Active Learning for Inference and Regeneration of Applications that Access Databases

JIASI SHEN and MARTIN C. RINARD, MIT EECS & CSAIL, USA

We present KONURE, a new system that uses active learning to infer models of applications that retrieve data from relational databases. KONURE comprises a domain-specific language (each model is a program in this language) and associated inference algorithm that infers models of applications whose behavior can be expressed in this language. The inference algorithm generates inputs and database contents, runs the application, then observes the resulting database traffic and outputs to progressively refine its current model hypothesis. Because the technique works with only externally observable inputs, outputs, and database contents, it can infer the behavior of applications written in arbitrary languages using arbitrary coding styles (as long as the behavior of the application is expressible in the domain-specific language). KONURE also implements a regenerator that produces a translated Python implementation of the application that systematically includes relevant security and error checks.

18

CCS Concepts: • **Software and its engineering → Source code generation**; **Domain specific languages**; **Software reverse engineering**;

Additional Key Words and Phrases: Active learning, program inference, program regeneration

ACM Reference format:
Jiasi Shen and Martin C. Rinard. 2021. Active Learning for Inference and Regeneration of Applications that Access Databases. *ACM Trans. Program. Lang. Syst.* 42, 4, Article 18 (January 2021), 119 pages.
https://doi.org/10.1145/3430952

1 INTRODUCTION

Progress in human societies is cumulative—each new generation builds on technology, knowledge, and experience accumulated over previous generations. Software collectively comprises one valuable store of human knowledge and experience as concretely realized in applications and software components. But there is currently no easy way to extract this knowledge and experience from its original context to productively deploy it into the new contexts that inevitably arise as societies evolve over time.

We present a new approach that uses *active learning* to infer models that capture the functionality of applications, specifically, the core functionality of the commands implemented in the target applications or components. These models comprise a mobile reification of the original functionality that can then be *regenerated* to obtain a new, clean version of the functionality specialized

This research was supported by DARPA (FA8650-15-C-7564) and Boeing (#Z0918-5060).
Authors' addresses: J. Shen and M. C. Rinard, MIT EECS & CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA; emails: {jiasi, rinard}@csail.mit.edu.

0164-0925/2021/01-ART18 $15.00
https://doi.org/10.1145/3430952

for immediate deployment into new languages, systems, or contexts. The regeneration can also improve the functionality by (1) discarding coding errors, (2) automatically inserting security and/or privacy checks into the regenerated code, and/or (3) improving the performance by applying optimizations appropriate for the new platform or context. In the longer term, active learning plus regeneration may also enable new development methodologies that work with simple prototype implementations as (potentially noisy) specifications, then use regeneration to automatically obtain clean, efficient implementations specialized for the specific context into which they will be deployed.

Applications that access databases are ubiquitous in computing systems. Such applications translate commands from the application domain into operations on the database, with the application constructing strings that it then passes to the database to implement the operations. Web servers, which accept HTTP commands from web browsers and interact with back-end databases to retrieve relevant data, are one particularly prominent example of such applications. These applications are written in a range of languages, often quickly become poorly understood legacy software, and, because they are typically directly exposed to Internet traffic, have been a prominent target for security attacks [13, 18, 37, 45, 46, 52, 53, 59]. Such applications therefore comprise a particularly compelling target for active learning plus regeneration.

1.1 Konure

We present a new system, Konure, that implements active learning plus regeneration for applications that retrieve data from relational databases. Konure systematically constructs database contents and application inputs, runs the application with the database and inputs, then observes the resulting database traffic and outputs to infer a model of application behavior.

Domain-Specific Language: To make the inference problem tractable, Konure works with a domain-specific language **(DSL)** that (1) captures common application behavior and (2) supports a hierarchical inference algorithm that progressively explores application behavior to infer the model. The inference algorithm (conceptually) maintains a current hypothesis as a sentential form of the grammar that defines the DSL. At each step it selects a nonterminal in this sentential form, constructs inputs and database contents that enable it to determine the one production to apply to this nonterminal that is consistent with the behavior of the application, configures the database, runs the application, then observes the resulting database traffic and outputs to refine the hypothesis by applying the inferred production to the nonterminal. Although we designed the DSL to be an internal representation that is invisible to users, it is straightforward to provide direct access to the DSL so users may write programs directly in the DSL.

The Black Box Approach: Konure treats the program as a black box, collecting only the externally observable behavior (inputs, outputs, and database traffic) of the program. This black box approach allows Konure to work directly with programs that would be difficult to analyze otherwise, such as programs that are obfuscated, built with complicated frameworks, or written in multiple languages.

Guarantees: If the application conforms to one of the models defined by the DSL, then the algorithm is guaranteed to (1) terminate and (2) produce an inferred program that correctly models the full core functionality of the application. Because Konure interacts with the application only via its inputs, outputs, and observed database interactions, it can infer and regenerate applications written in any language or in any coding style or methodology.

Benefits: Because the model captures core application functionality, it can help developers explore and better understand this functionality. Konure can also regenerate the application into a potentially different language and systematically apply coding patterns and additional checks that are known to be safe. Konure therefore targets several use cases: (1) security and/or performance through safe regenerated code, (2) portability to new platforms, (3) reverse engineering, and (4) program understanding.

1.2 Key Inferrability Properties

The design of the KONURE DSL, together with its associated top-down inference algorithm, is a central contribution of this article. We next outline several key properties of the design that enable inferrability via active learning.

In general, programs contain statements linked together by control and data flow. To promote control-flow inferrability, each statement in the DSL executes a query that is directly observable in the intercepted database traffic. All control flow is tied directly to the query results—If statements test if their query retrieves empty data; For statements iterate over all rows that their query retrieves, with all iterations independent. These properties help KONURE generate a focused, tractably small sequence of inputs and database contents that (1) finds and traverses all relevant control-flow paths and (2) completely resolves each For loop with a single execution of two or more iterations.

To promote data flow inferrability, all data flows directly from either input parameters or retrieved query results to executed queries or outputs. KONURE infers the data flow by matching concrete values in executed queries or outputs against the input parameter or retrieved query result with the same value. KONURE eliminates potential data flow ambiguities by populating the input parameters and database contents with appropriately distinct concrete values.

The DSL is designed to enable the formulation of all properties of interest as quantifier-free SMT formulas. KONURE leverages this property to construct inputs and databases that explore all relevant control-flow paths and deliver the distinct values that enable KONURE to infer the data flow.

1.3 Experimental Results

We present case studies applying KONURE to five applications: Fulcrum Task Manager [2], Kandan Chat Room [4], Enki Blogging Application [1], Blog [3], and a student registration application developed by an independent evaluation team to test SQL injection attack detection and nullification techniques. Our results show that KONURE is able to successfully infer and regenerate commands that these applications use to retrieve data from the database.

1.4 Contributions

This article makes the following contributions:

- **Inference Algorithm:** It presents a new algorithm for inferring the behavior of database-backed applications. Conceptually, the algorithm works with hypotheses represented as sentential forms of the grammar of KONURE DSL. At each stage the algorithm systematically constructs database contents and application inputs, runs the application, and observes the resulting database traffic and outputs to resolve a selected nonterminal in the current hypothesis. This approach enables KONURE to work effectively with unbounded model spaces to infer models that capture the core functionality of the target class of applications.
- **DSL Design:** It presents a DSL for capturing specific computational patterns typically implemented by database-backed applications. The inference algorithm and DSL are designed together to enable an effective active learning algorithm that leverages the structure of the DSL to iteratively refine hypotheses represented as sentential forms in the DSL grammar.

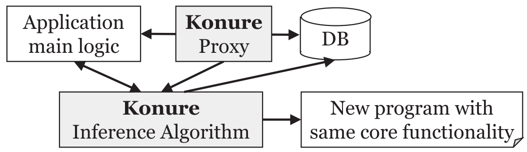

Fig. 1. The Konure architecture, including a transparent proxy interposed between the application and the database to observe the generated database traffic.

- **Soundness and Completeness:** It presents a key theorem that states that if the behavior of the application conforms to the DSL, then the inference algorithm infers a program that correctly captures the full core functionality of the application.
- **Regeneration:** It shows how to regenerate new versions of the application that implement safe computational patterns and contain appropriate safety and security checks. The regenerator encapsulates the knowledge required to work effectively in the target domain and can eliminate coding errors that lead to incorrect application behavior or security vulnerabilities.
- **Experimental Results:** It presents results using Konure to infer and regenerate commands written in Ruby on Rails and Java. The results highlight Konure's ability to infer and regenerate robust, safe Python implementations of commands originally coded in other languages.

2 EXAMPLE

We next present an example that illustrates how Konure infers and regenerates a database-backed application. The example is a student registration system adapted from an application written by an independent evaluation team hired by an agency of the United States government to evaluate techniques for detecting and nullifying SQL injection attacks. The application was written in Java and interacts with a MySQL database [84] via JDBC [65].

Command: The application implements the following command: "`liststudentcourses -s` *s* `-p` *p*", where the input parameter *s* denotes student ID and *p* denotes password. The application first checks whether the student with ID *s* has password *p* in the database. If so, the application displays the list of courses for which this student has registered, along with the teacher for each course.

Database: The database contains: (1) a `student` table, which contains student ID (primary key), first name, last name, and password; (2) a `teacher` table, which contains teacher ID (primary key), first name, and last name; (3) a `course` table, which contains course ID (primary key), name, course number, and teacher ID; and (4) a `registration` table, which contains student ID and course ID.

First Execution: The Konure inference algorithm configures an empty database, then executes the application with the command "`liststudentcourses -s 0 -p 1`," which sets input parameters *s* and *p* to 0 and 1, respectively. Konure uses a transparent proxy (Figure 1) to observe the resulting database traffic, which the proxy collects as the *concrete trace* of the execution (Figure 3(a)). The query uses the constant `'0'`, which comes from the input parameter *s*, and retrieves no data from the (empty) database. For this execution, the application produces no output.

Based on this information, Konure rewrites the concrete trace to replace concrete values (such as `'0'`) with *origin locations*, which identify the source of each value. The result is a corresponding *abstract trace* (Figure 3(b)). This abstract trace contains a query `q1` that selects all columns from the `student` table. The selection criterion is that the student ID must equal the input parameter *s*. Konure derives the origin locations by matching concrete values in the concrete trace against input values and values in the database.

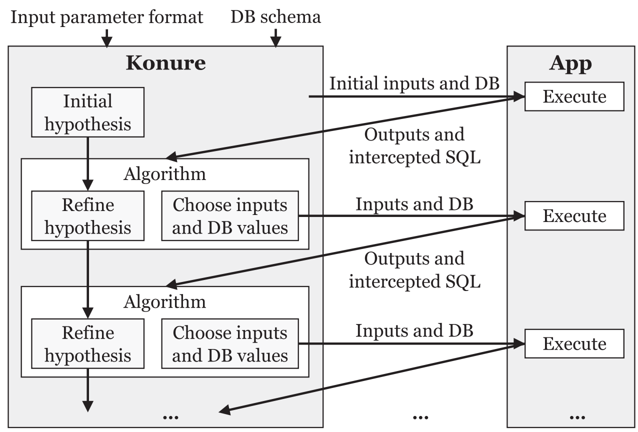

Fig. 2. The Konure active learning algorithm iteratively refines its hypothesis to infer the application.

```
SELECT * FROM student WHERE id = '0'
```

(a) Concrete trace from the first execution. The database is empty and the query retrieves zero rows.

```
q1: select student.id, student.password, student.firstname, student.lastname
    where student.id = s
```

(b) Abstract trace from the first execution, converted from the concrete trace in Figure 3a. The conversion replaces the constant `'0'` with its origin location, the input parameter s.

Fig. 3. First execution trace.

$$
\begin{array}{lcl}
\text{Prog} & := & \epsilon \mid \text{Seq} \mid \text{If} \mid \text{For} \\
\text{Seq} & := & \text{Query Prog} \\
\text{If} & := & \texttt{if}\ \text{Query}\ \texttt{then}\ \text{Prog}\ \texttt{else}\ \text{Prog} \\
\text{For} & := & \texttt{for}\ \text{Query}\ \texttt{do}\ \text{Prog}\ \texttt{else}\ \text{Prog} \\
\text{Query} & := & y \leftarrow \texttt{select}\ \text{Col}^{+}\ \texttt{where}\ \text{Expr}\,;\ \texttt{print}\ \text{Orig}^{*} \\
\text{Expr} & := & \texttt{true} \mid \text{Expr} \wedge \text{Expr} \mid \text{Col} = \text{Col} \mid \text{Col} = \text{Orig} \\
\text{Col} & := & t.c \\
\text{Orig} & := & x \mid y.\text{Col}
\end{array}
$$

$$x, y \in \mathit{Variable}, \quad t \in \mathit{Table}, \quad c \in \mathit{Column}$$

Fig. 4. Grammar for the Konure DSL.

Konure DSL: Figure 4 presents the (abstract) grammar for the Konure DSL. A program consists of a sequence of Query statements potentially terminated by an If or For statement. An If statement does not test an arbitrary condition—it instead only tests if the Query in the condition retrieves empty or nonempty data. Similarly, a For statement does not iterate over an arbitrary list—it instead iterates over the rows in its Query, executing its `else` clause if its Query retrieves zero rows. These restrictions (among others, Section 3.1) are key to the inferrability of the DSL.

First Production: The first execution generated a single query (Figure 3(a)). Konure determines if this query came from a Seq, If, or For statement as follows: Working with the abstract trace in Figure 3(b), Konure generates three sets of constraints. Each set specifies input parameters and database contents. The first set specifies that the query retrieves zero rows. The second specifies that the query retrieves at least one row. The third specifies that the query retrieves at least two

```
SELECT * FROM student WHERE id = '5'
SELECT * FROM student WHERE id = '5' AND password = '6'
```

(a) Concrete trace from the second execution. The context is configured to ensure that the first query retrieves at least one row.

```
q1: select student.id, student.password, student.firstname, student.lastname
    where student.id = s
q2: select student.id, student.password, student.firstname, student.lastname
    where student.id = s ∧ student.password = p
```

(b) Abstract trace from the second execution, converted from the concrete trace in Figure 5a. The conversion replaces the constants '5' and '6' with their origin locations, input parameters s and p.

Fig. 5. Second execution trace.

```
if y1 ← select student.id, student.password, student.firstname, student.lastname
    where student.id = s then P1 else P2
```

Fig. 6. Hypothesis after resolving the topmost Prog nonterminal to an If statement.

rows. Konure invokes an SMT solver to obtain a *context* for each set of constraints. Each context identifies inputs and database values that satisfy the constraints.

In the example the third set of constraints is unsatisfiable, because the query accesses the primary key and there is at most one row for each value of the primary key. The first and second sets of constraints are satisfiable and therefore produce viable contexts. Konure executes the application in each of these contexts. Figures 3(a) and 5(a) present the recorded concrete traces; Figures 3(b) and 5(b) present the corresponding abstract traces. These traces indicate that the observable behavior of the application differs depending on whether the first Query retrieves no rows (Figures 3(a) and 3(b)) or at least one row (Figures 5(a) and 5(b)). Konure concludes that the first Query comes from an If statement and produces the first hypothesis in Figure 6. This hypothesis corresponds to applying an If production to the topmost Prog nonterminal.

Intuition: Recall that, in the Konure DSL (Figure 4), there are four potential productions to apply for each Prog nonterminal: Prog := ϵ, Prog := Seq, Prog := If, and Prog := For. Konure resolves each Prog nonterminal in turn by applying the appropriate production. For the topmost Prog nonterminal, applying the ϵ production would result in an empty program, which is incorrect, because the program has produced nonempty traces (Figures 3 and 5). Applying the Seq production would result in a program that does not condition on the results of the first query (q1 in Figures 3(b) and 5(b)), which is incorrect, because the program behavior differs in the first two traces—the program terminates immediately after the first query in the first trace (Figure 3) but continues execution after the first query in the second trace (Figure 5). Applying the For production would result in a program that iterates over the rows retrieved by the first query, which is incorrect, because the query retrieves at most one row, making iterations unobservable. The If production is the only production that is consistent with the observed program behavior.

Second Production: Konure next resolves the P_1 nonterminal in the first hypothesis. Working with the abstract trace in Figure 5(b), Konure generates three sets of constraints that (1) force the first query (q1) to retrieve at least one row (this constraint forces the application to execute the `then` branch of the topmost If statement) and (2) force the second query (q2) to retrieve no rows, at least one row, and at least two rows, respectively. Once again, the first two sets of constraints produce viable contexts; the third is unsatisfiable.

Figure 5(a) presents the trace from the execution in which the second query retrieves no rows; Figure 7(a) presents the trace from the execution in which the second query retrieves at least

```
SELECT * FROM student WHERE id = '1'
SELECT * FROM student WHERE id = '1' AND password = '2'
SELECT * FROM course c JOIN registration r ON r.course_id = c.id WHERE r.
    student_id = '1'
```

(a) Concrete trace from the third execution. The context is configured so that the first and second queries retrieve at least one row and the third query retrieves zero rows.

```
q1: select student.id, student.password, student.firstname, student.lastname
    where student.id = s
q2: select student.id, student.password, student.firstname, student.lastname
    where student.id = s ∧ student.password = p
q3: select course.id, course.name, course.course_number, course.size_limit, course.
    is_offered, course.teacher_id, registration.student_id, registration.course_id
    where registration.course_id = course.id ∧ registration.student_id = s
```

(b) Abstract trace from the third execution, converted from the concrete trace in Figure 7a. The conversion replaces the constants '1' and '2' with their origin locations, input parameters s and p.

Fig. 7. Third execution trace.

```
if y1 ← select student.id, student.password, student.firstname, student.lastname
    where student.id = s then {
  if y2 ← select student.id, student.password, student.firstname, student.lastname
      where student.id = s ∧ student.password = p
  then P3 else P4 } else P2
```

Fig. 8. Hypothesis after resolving P_1 (Figure 6).

one row. Because the traces differ (similarly to the above First Production), Konure resolves the nonterminal P_1 to an If statement. Figure 8 presents the resulting hypothesis.

Intuition: As with the topmost Prog nonterminal, the P_1 nonterminal (Figure 6) also has four potential productions: Prog := ϵ, Prog := Seq, Prog := If, and Prog := For. For P_1, applying the ϵ production would result in a program with an empty `then` branch (Figure 6), which is incorrect, because the program has produced traces that perform actions after the first query (q1 in Figures 5(b) and 7(b)) retrieves nonempty data (Figures 5 and 7). Applying the Seq production to P_1 would result in a program that does not condition on the results of the first query in P_1 (q2 in Figures 5(b) and 7(b)), which is incorrect, because the program behavior differs in the second and the third traces—the program terminates immediately after the second query in the second trace (Figure 5) but continues execution after the second query in the third trace (Figure 7). Applying the For production to P_1 would result in a program that iterates over the rows retrieved by the first query in P_1, which is incorrect, because the query retrieves at most one row, making iterations unobservable. The If production is the only production that is consistent with the observed program behavior.

Third Production: Konure next resolves the P_3 nonterminal. Working with the abstract trace produced by the previous step (Figure 7(b)), Konure generates constraints that force the application to execute P_3, once again with zero, at least one, or at least two rows retrieved by the first query in P_3 (q3 in Figure 7(b)). The solver generates viable contexts for all three sets of constraints. For the context with at least two rows retrieved, Konure collects the trace in Figure 9.

In this execution the third query retrieves two rows. The Konure loop detection algorithm examines the trace, detects the repetitive pattern in the last four queries, concludes that the application iterates over all of the rows retrieved from the third query, and resolves P_3 to a For statement.

```
SELECT * FROM student WHERE id = '3'
SELECT * FROM student WHERE id = '3' AND password = '4'
SELECT * FROM course c JOIN registration r ON r.course_id = c.id WHERE r.
    student_id = '3'
SELECT firstname, lastname FROM teacher WHERE id = '16'
SELECT count(*) FROM registration WHERE course_id = '12'
SELECT firstname, lastname FROM teacher WHERE id = '11'
SELECT count(*) FROM registration WHERE course_id = '7'
```

Fig. 9. Concrete trace from an execution to resolve P_3 (Figure 8). The third query retrieves two rows. The final four queries are generated by a loop that iterates over the retrieved two rows.

```
if y1 ← select student.id, student.password, student.firstname, student.lastname
    where student.id = s then {
 if y2 ← select student.id, student.password, student.firstname, student.lastname
     where student.id = s ∧ student.password = p
 then {
  for y3 ← select course.id, course.name, course.course_number, course.size_limit,
      course.is_offered, course.teacher_id, registration.student_id, registration.
      course_id
      where registration.course_id = course.id ∧ registration.student_id = s;
      print y3.course.id, y3.course.teacher_id
  do P5 else P6 } else P4 } else P2
```

Fig. 10. Hypothesis after resolving P_3 (Figure 8).

For this execution the application also produces the `id` and `teacher_id` columns from the retrieved rows of the `course` table as output. The updated hypothesis (Figure 10) therefore contains a Print statement that prints these values.

Intuition: As with the previous Prog nonterminals, the P_3 nonterminal (Figure 8) also has four potential productions: Prog := ϵ, Prog := Seq, Prog := If, and Prog := For. For P_3, applying the ϵ production would result in a program with an empty inner `then` branch (Figure 8), which is incorrect, because the program has produced traces that perform actions after the second query (q2 in Figure 7(b)) retrieves nonempty data (Figures 7 and 9). Applying the Seq production to P_3 would result in a program that does not condition on the results of the first query in P_3 (q3 in Figure 7(b)), which is incorrect, because the program behavior differs in the third and the fourth traces—the program terminates immediately after the third query in the third trace (Figure 7) but continues execution after the third query in the fourth trace (Figure 9). The remaining two potential productions are If and For. To choose the appropriate production, Konure obtains an execution where the third query retrieves at least two rows (Figure 9). In this execution, the third query retrieves two rows, followed by two repetitions of a set of two queries. Because the row count matches the repetition count, the trace is consistent with a potential For statement that iterates over the rows retrieved by the third query. We designed the Konure DSL to restrict certain repetitive queries in the program (more details in Section 3.1), so this repetition is plausible only when P_3 resolves to a For statement.

Regeneration: Konure proceeds as above, systematically targeting and resolving nonterminals in the hypothesis, until all of the nonterminals are resolved and it has inferred a model of the command. It can then regenerate the command, inserting security/safety checks as desired. Our current Konure implementation regenerates Python code using a standard SQL library to perform the database queries. This regeneration eliminates a seeded SQL injection attack vulnerability present in the original program.

Noisy Specifications: Because the active learning algorithm, guided by the DSL, tends to generate contexts that conform to common use cases, Konure can work productively with programs that contain obscure corner-case bugs not exercised during the inference [54, 66]. The SQL injection attack vulnerability present in the original Student Registration application but discarded in the regeneration is an example of just such an obscure corner case bug. We view such programs as *noisy specifications*. Given the known challenges developers face when attempting to deliver correct programs, we consider the ability of Konure to work successfully with such noisy specifications as a significant advantage of the overall approach.

Developer Understanding: In a deployed system, we expect that developers would be given examples and documentation that outline the Konure DSL and the model of computation. We expect that this information, along with experience using Konure, would enable developers to work productively with Konure using programs written in their language of choice.

3 DESIGN

Inference Algorithm Overview: Konure infers two aspects of the program. First, starting from a concrete trace intercepted by the proxy (Figure 1), Konure locates the concrete values and infers their origin locations. To infer the origin locations, Konure keeps track of the concrete values that are available when the program performs each SQL query. To disambiguate different origin locations that happen to hold the same concrete value in an execution, Konure adopts a demand-driven approach. With the origin locations inferred, Konure constructs an abstract trace.

Second, starting from the unstructured sequences of queries in traces, Konure infers the underlying control flow in the program. This inference algorithm is constructive [12]—instead of enumerating candidate solutions, the algorithm constructs the solution progressively every time Konure finds an interesting behavior of the application. During inference, Konure maintains a hypothesis of what is currently known about the program. The hypothesis is (conceptually) represented as a sentential form in the Konure DSL, with nonterminals denoting hidden parts that are left to infer. The algorithm starts with a Prog nonterminal as its initial hypothesis, then progressively resolves Prog nonterminals until it completely infers the program. The algorithm resolves each of the Prog nonterminals by applying an appropriate production, that is, applying one of Prog := ϵ, Prog := Seq, Prog := If, and Prog := For (Figure 4). Konure chooses the appropriate production based on three (potential) executions of the program, forcing a specific query to retrieve zero rows, at least one row, and at least two rows, respectively. These three executions are sufficient for Konure to uniquely determine the correct production for the current Prog nonterminal. The inference proceeds by expanding nonterminals until it obtains a complete program. As Konure recursively traverses the paths through the program as expressed in the DSL, it maintains path constraints that lead to the next part of the program to infer. Instead of maintaining the current hypothesis as an explicit sentential form, Konure represents the hypothesis implicitly in the data structures and recursive structure of the inference algorithm as it executes.

3.1 Konure Domain-specific Language

Konure infers application functionality that can be expressed in the Konure DSL.

3.1.1 DSL Overview. We design the Konure DSL to precisely capture the programs that our technique works with. A goal here is to balance between expressiveness and inferrability. We outline the expressiveness in this section and defer the discussion on inferrability to Sections 3.1.3 and 3.1.4.

The Konure DSL captures a range of data retrieval applications that work with an external database. A user runs an application through interfaces such as command line arguments or HTTP requests. When the application runs, it sends SQL queries to the database, which retrieves data as requested. For many real-world applications, such as forums, blogs, and inventory management systems, a significant part of their core functionality is dedicated to retrieving data in this form. In practice, many of these applications have multiple commands that access different parts of the database. In this research, we infer one command at a time, and we refer to each command as a program.

Many of these programs share an interesting pattern: The data flow often manifests as SQL queries, and the control flow largely depends on the query results. As an example of conditional statements that depend on query results, a program may first look up a user's name in the database and then execute one of two branches, that is, (1) if the user does not exist, then print an error and terminate or (2) if the user exists, then perform more queries to look up more information. As an example of loops that depend on query results, a program may first retrieve a list of articles in the database, then repeatedly perform the same action on each of these articles. When a program's control flow largely depends on the database queries, the database traffic during program execution may reveal much information about the program functionality. The Konure DSL is designed to capture data retrieval programs that have this common pattern.

3.1.2 DSL Definition. We present the grammar for the Konure DSL in Figure 4. Each query in this DSL performs an SQL `select` operation that retrieves data from specified columns in specified tables. Our current DSL supports SQL `where` clauses that select rows in which one column has the same value as another column (Col = Col) or the same value as a value in the context (Col = Orig). Selecting from multiple tables corresponds to an SQL join operation. The query stores the retrieved data in a unique variable (y) for later use. All variables must be defined before they are used.

The control flow in the DSL is directly tied to queries and their results. An If statement first performs a query to retrieve data. If the query retrieves nonempty data, it enters the `then` branch, otherwise the `else` branch. A For statement likewise first performs a query. If the query retrieves nonempty data, the loop body executes once for each row retrieved by the query. If the query retrieves empty data, execution enters the `else` branch.

To enable the Konure inference algorithm to effectively distinguish If statements from Seq statements, Konure requires the two branches of each If statement to start with queries that have different skeletons (or one of the branches must be empty). To facilitate effective loop detection, Konure requires the first query after any query that may retrieve multiple rows to have a skeleton that is distinct from all subsequent queries. Konure also requires that the program have no nested loops.

The outcomes of executing a program consists of a concrete trace, which consists of the intercepted SQL queries, and the output values produced by Print statements. Each Print statement is associated with a query and only prints values retrieved by its query.

To formally define the Konure DSL, we first define skeletons (Definition 3.1), then describe the externally observable part of a program (Definition 3.2), and finally define the DSL as a subset of programs in Prog (Figure 4) that satisfy two additional restrictions (Definition 3.3 and Definition 3.4). For readability, we present Definition 3.2 and Definition 3.3 here only at a high level and postpone their formalization to Section 4.

Definition 3.1. The *skeleton* of a program $P \in$ Prog is a program that is syntactically identical to P except for replacing syntactic components derived from the Orig nonterminal (Figure 4) with an empty placeholder $\diamond$. For reference, we present the syntax for skeleton programs in Appendix A. We write $\mathcal{S}$ for the set of skeleton programs. For readability, we shall denote Prog elements (Figure 4) with uppercase letters, except for variables such as x, y, and denote $\mathcal{S}$ elements with lowercase letters.

We shall write $\pi_{\mathcal{S}} P$ for the skeleton of program $P \in \text{Prog}$; it is defined as follows:

$$\begin{aligned}
\pi_{\mathcal{S}} \epsilon &= \epsilon \\
\pi_{\mathcal{S}} (Q\ P) &= \pi_{\mathcal{S}} Q\ \pi_{\mathcal{S}} P \\
\pi_{\mathcal{S}} (\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2) &= \texttt{if}\ \pi_{\mathcal{S}} Q\ \texttt{then}\ \pi_{\mathcal{S}} P_1\ \texttt{else}\ \pi_{\mathcal{S}} P_2 \\
\pi_{\mathcal{S}} (\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2) &= \texttt{for}\ \pi_{\mathcal{S}} Q\ \texttt{do}\ \pi_{\mathcal{S}} P_1\ \texttt{else}\ \pi_{\mathcal{S}} P_2 \\
\pi_{\mathcal{S}} (y \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ E\ ;\ \texttt{print}\ \overline{O}) &= \diamond \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ \pi_{\mathcal{S}} E\ ;\ \texttt{print}\ \diamond \\
\pi_{\mathcal{S}} \texttt{true} &= \texttt{true} \\
\pi_{\mathcal{S}} (E_1 \wedge E_2) &= \pi_{\mathcal{S}} E_1 \wedge \pi_{\mathcal{S}} E_2 \\
\pi_{\mathcal{S}} (C_1 = C_2) &= (C_1 = C_2) \\
\pi_{\mathcal{S}} (C = O) &= (C = \diamond),
\end{aligned}$$

where $P, P_1, P_2 \in \text{Prog}$, $Q \in \text{Query}$, $C, C_1, C_2 \in \text{Col}$, $E, E_1, E_2 \in \text{Expr}$, $O \in \text{Orig}$, and $y \in \mathit{Variable}$. We use an overline, as in $\overline{C}$ and $\overline{O}$, to denote a list.

Clearly, for any program $P \in \text{Prog}$, we have $\pi_{\mathcal{S}} P \in \mathcal{S}$.

Definition 3.2. For any program $P \in \text{Prog}$, $\widetilde{P}$ is the semantically equivalent program obtained from P by discarding unreachable branches in If and For statements, downgrading For statements with empty loop bodies or loop bodies that execute at most once to If statements, and downgrading If statements with an unreachable branch or two semantically equivalent branches to Seq statements. We present the algorithms for this code transformation in Section 4.1.

Definition 3.3. For any program $P \in \text{Prog}$, $\mathcal{T}(P)$ is the set of queries in P that retrieve at least two rows in some execution.[1] $\mathcal{R}(P)$ is the set of all queries Q in P with two subsequent queries Q_1 and Q_2 such that Q_1 immediately follows Q in the program, Q_1 does not appear as the first query of an `else` branch of an If or For statement, Q_2 occurs after Q_1 in the program, and Q_1 and Q_2 have the same skeleton. $\mathcal{D}(P)$ is a predicate that is `true` if and only if the two branches of all conditional statements in P start with queries with different skeletons (or one of the branches is empty). We formally define the functions $\mathcal{T}(\cdot)$, $\mathcal{R}(\cdot)$, and $\mathcal{D}(\cdot)$ in Section 4.2.

Definition 3.4 (The Konure DSL). We define the Konure DSL as the set of programs $\mathcal{K} \subset \text{Prog}$ defined as:

$$\mathcal{K} = \{\widetilde{P} \mid P \in \text{Prog}, \mathcal{T}(\widetilde{P}) \cap \mathcal{R}(\widetilde{P}) = \emptyset, \mathcal{D}(\widetilde{P}) = \texttt{true}\}.$$

The first restriction, $\mathcal{T}(\widetilde{P}) \cap \mathcal{R}(\widetilde{P}) = \emptyset$, states that if a query may retrieve multiple rows from the database, then the next query does not share a skeleton with any other subsequent query in the program. This restriction facilitates loop detection by eliminating repeated queries that do not come from iterations of the same loop (Section 3.2.2).[2] The second restriction, $\mathcal{D}(\widetilde{P}) = \texttt{true}$, states that the two branches of any If statement in $\widetilde{P}$ must start with queries with different skeletons (or one of the branches must be empty). Intuitively, this restriction enables Konure to efficiently distinguish Seq from If statements (Section 4).

We present several immediate extensions to $\mathcal{K}$ in Section 5.

Because of the focused expressive power of the Konure DSL, it is possible to decide all relevant conditions statically, rewrite P to $\widetilde{P}$, and determine if $\widetilde{P} \in \mathcal{K}$. Note that because programs $P \in \mathcal{K}$ may reference values using distinct but semantically equivalent variables, $\mathcal{K}$ is not a true canonical

[1] A query will never retrieve more than one row if, for example, it selects rows that have a specific primary key value.

[2] Our implemented Konure prototype deploys a more sophisticated loop detection algorithm that enables it to relax this restriction.

form, i.e., there may be distinct but semantically equivalent programs in $\mathcal{K}$. It is possible, however, to eliminate such equivalences by replacing each variable with the first semantically equivalent variable to occur in the program. This transformation is implementable with an SMT solver and eliminates distinct but semantically equivalent programs to deliver a true canonical form for the Konure DSL.

3.1.3 Design Rationale. The DSL captures a wide range of applications that display data from a database by retrieving data based on inputs and database contents. Meanwhile, these applications are restrictive enough to be inferred efficiently.

Because the DSL directly ties the control flow to query results, Konure can effectively observe the control flow execution by observing the database traffic (Figure 1). For example, the student registration program in Section 2 enters two different branches in the first two executions, which is inferrable by comparing the two corresponding traces (Figures 3 and 5).

Another benefit of tying the control flow to query results in the DSL is that Konure can effectively force the program to execute down certain paths. Konure achieves this goal by carefully choosing values for the inputs and the database so, when Konure executes the program, the relevant queries retrieve appropriate numbers of rows that lead to the path. For example, to force the student registration program (Section 2) to enter a (potential) branch unvisited by the first execution (Figure 3), Konure solves for a set of new input and database values to guarantee that, in the second execution, the first query retrieves at least one row (Figure 5).

3.1.4 Expressiveness and Limitations. Konure works well with programs whose behavior conforms to the Konure DSL, though the programs themselves can be implemented in any language or in any coding style or methodology. Two key properties of the Konure DSL are that (1) the data flow manifests as database queries, which are directly observable in the database traffic, and (2) the control flow is directly tied to the query results. Konure takes advantage of these properties to actively explore various paths in the program. The outcome is an accurate inferred model of the program, and the inference algorithm does not require an analysis of the source code or the binary of the program.

The Konure inference algorithm may not extend well to infer unknown conditional expressions or arithmetic calculations in the program that are not directly observable. Example programs include online-shopping applications, whose core functionality often involve numeric calculations that are not implemented as database query expressions. We discuss unsupported programs in Section 6.2. In general, Konure is not designed to infer programs that cannot be captured by an inferrable DSL.

However, it is straightforward to extend Konure to support applications with SQL queries that involve relational comparisons (besides equality and membership checks), simple arithmetics, constants, or aggregate functions. It is straightforward because (1) we use an SMT solver that supports solving constraints involving these operators, and (2) the operators are directly present in the intercepted SQL queries. Because experience with SMT solvers in other contexts shows that these solvers readily support formulas with these kinds of operators and constraints, we do not anticipate any significant performance issues with this extension. It is also possible to extend Konure to access not only the database traffic, but also other runtime or descriptive information of the program. For example, one could first statically extract all of the constant values used in the program (binary or source code), then take advantage of these known constants while inferring conditional checks. Another way to extend Konure is to incorporate domain knowledge about computations that are well known, widely used, and easy to reason about in the solver. Example computations include standard string manipulations (such as concatenation, splitting, and capitalizing), date and time conversions, and number translations. We anticipate that adding these features would require only relatively small changes to the overall framework of the inference algorithm.

3.2 Konure Inference Algorithm

We present the Konure inference algorithm (Algorithm 1) for a program P that implements a single command. For programs with multiple commands, Konure uses Algorithm 1 to infer each command in turn.

Recall that, conceptually, the Konure inference algorithm works with hypotheses represented as sentential forms of the DSL grammar. The algorithm systematically constructs inputs and database contents, runs the program, and observes the resulting database traffic and outputs to resolve a selected nonterminal in the current hypothesis.

Algorithm 1 configures an empty database, sets the parameters to distinct values, invokes Algorithm 2 to run the program and obtain an initial trace, then invokes Algorithm 6 to recursively infer the program. The inference algorithm works with *deduplicated* annotated traces t that record one iteration of each executed loop, so the structure of the trace matches the corresponding path through the program.

3.2.1 Notation. Before presenting the algorithms, we first define the relevant terminologies and notation, including contexts, origin locations, concrete traces, abstract traces, annotated traces, and path constraints.

Definition 3.5. A context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \mathit{Context}$ contains value mappings for the input parameters ($\sigma_I \in \mathit{Input}$), database contents ($\sigma_D \in \mathit{Database}$), and results retrieved by database queries ($\sigma_R \in \mathit{Result}$):

$$\begin{aligned} \sigma \in \mathit{Context} &= \mathit{Input} \times \mathit{Database} \times \mathit{Result} \\ \sigma_I \in \mathit{Input} &= \mathit{Variable} \rightarrow \mathit{Value} \\ \sigma_D \in \mathit{Database} &= \mathit{Table} \rightarrow \mathbb{Z}_{>0} \rightarrow \mathit{Column} \rightarrow \mathit{Value} \\ \sigma_R \in \mathit{Result} &= \mathit{Variable} \rightarrow \mathbb{Z}_{>0} \rightarrow \mathit{Table} \rightarrow \mathit{Column} \rightarrow \mathit{Value} \\ \mathit{Value} &= \mathit{Int} \cup \mathit{String}. \end{aligned}$$

The input context σ_I maps input parameter variables $x \in \mathit{Variable}$ to concrete values. The database context σ_D maps database locations (identified by a table name, a row number, and a column name) to concrete values. The results context σ_R maps each query result variable $y \in \mathit{Variable}$ to a list of rows, with each value in each row identified by the table and column from which it was retrieved.

Example 1. In Section 2, the first execution of the program (Figure 3) uses the following context:

$$\sigma_1 = \langle \{s : \text{‘}\mathtt{0}\text{’}, p : \text{‘}\mathtt{1}\text{’}\}, \{\mathtt{student} : \emptyset, \mathtt{teacher} : \emptyset, \mathtt{course} : \emptyset, \mathtt{registration} : \emptyset\}, \emptyset \rangle.$$

This context sets input parameters s and p to 0 and 1, respectively, and sets all database tables to empty. The second execution (Figure 5) uses the following context:

$$\begin{aligned} \sigma_2 = \langle &\{s : \text{‘}\mathtt{5}\text{’}, p : \text{‘}\mathtt{6}\text{’}\}, \\ &\{\mathtt{student} : \{1 : \{\mathtt{id} : \text{‘}\mathtt{5}\text{’}, \mathtt{password} : \text{‘}\mathtt{2}\text{’}, \mathtt{firstname} : \text{‘}\mathtt{3}\text{’}, \mathtt{lastname} : \text{‘}\mathtt{4}\text{’}\}\}, \\ &\ \mathtt{teacher} : \emptyset, \mathtt{course} : \emptyset, \mathtt{registration} : \emptyset\}, \emptyset \rangle. \end{aligned}$$

This context sets input parameters s and p to ‘`5`’ and ‘`6`’, respectively, and sets the `student` table to contain one row whose column `id` equals the input s.

Definition 3.6. An *origin location* $O \in \mathrm{Orig}$ in a program $P \in \mathrm{Prog}$ is an occurrence of a variable x or a column $y.\mathrm{Col}$ in a query result y.

Definition 3.7. For an origin location $O \in \text{Orig}$ and a context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \mathit{Context}$, $\sigma(O)$ denotes the result from looking up O in σ. Specifically, for an input parameter $x \in \mathit{Variable}$, $\sigma(x) = \sigma_I(x)$. For a query result variable $y \in \mathit{Variable}$, table $t \in \mathit{Table}$, and column $c \in \mathit{Column}$, $\sigma(y.t.c) = \sigma_R(y)(t)(c)$. When a program references a query result variable that holds multiple rows, they are referenced as a list.

For a query $Q \in \text{Query}$, $\mathit{SQL}_\sigma(Q)$ denotes the concrete query Q in SQL syntax:

$$\begin{aligned}
\mathit{SQL}_\sigma(y \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ E\ ;\ \texttt{print}\ \overline{O}) &= \texttt{SELECT}\ \overline{C}\ \texttt{FROM}\ \mathit{Join}(\overline{C}, E)\ \texttt{WHERE}\ \mathit{SQL}_\sigma(E) \\
\mathit{SQL}_\sigma(\texttt{true}) &= \texttt{true} \\
\mathit{SQL}_\sigma(E_1 \land E_2) &= \mathit{SQL}_\sigma(E_1)\ \texttt{AND}\ \mathit{SQL}_\sigma(E_2) \\
\mathit{SQL}_\sigma(C_1 = C_2) &= (C_1 = C_2) \\
\mathit{SQL}_\sigma(C = O) &= \begin{cases} C = \sigma(O) & \text{if } \sigma(O) \text{ is a value} \\ C\ \texttt{IN}\ \sigma(O) & \text{if } \sigma(O) \text{ is a list,} \end{cases}
\end{aligned}$$

where $C, C_1, C_2 \in \text{Col}$, $E, E_1, E_2 \in \text{Expr}$, $O \in \text{Orig}$, and $y \in \mathit{Variable}$. We use an overline, as in $\overline{C}$ and $\overline{O}$, to denote a list. The $\mathit{Join}(\overline{C}, E)$ operation collects the relevant tables in $\overline{C}$ to construct corresponding SQL `JOIN` operations, using the checks in E to construct the relevant `ON` expressions.

$\sigma(Q)$ denotes the result from evaluating Q in σ. Evaluating Q involves replacing origin locations in Q with their values in σ_I and σ_R, rewriting the query in SQL syntax ($\mathit{SQL}_\sigma(Q)$), then performing the SQL query on σ_D. The query result contains an ordered list of rows. $|\sigma(Q)|$ denotes the number of rows in $\sigma(Q)$. $Q.y$ denotes the variable that stores the retrieved data. $\sigma[Q.y \mapsto z]$ denotes the new context after updating σ_R to map $Q.y$ to z. When the new content z is empty, we shall write σ for $\sigma[Q.y \mapsto \emptyset]$.

$\mathit{Print}_\sigma(Q)$ denotes the output from Q: if Q is of the form "$y \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ E\ ;\ \texttt{print}\ \overline{O}$", then $\mathit{Print}_\sigma(Q) = \overline{\sigma[y \mapsto \sigma(Q)](O)}$, where $C \in \text{Col}$, $E \in \text{Expr}$, and $O \in \text{Orig}$.

Example 2. Following the notation in Example 1, we have $\sigma_1(s) = $ '0', $\sigma_1(p) = $ '1', $\sigma_2(s) = $ '5', and $\sigma_2(p) = $ '6'. Let Q_1 be the first inferred query in Figure 6, that is,

$$\begin{aligned}
Q_1 = y_1 \leftarrow\ & \textbf{select}\ \texttt{student.id}, \texttt{student.password}, \texttt{student.firstname}, \texttt{student.lastname} \\
& \textbf{where}\ \texttt{student.id} = s\ ;\ \textbf{print}\ \texttt{[]}.
\end{aligned}$$

We have concrete queries[3]:

$$\begin{aligned}
\mathit{SQL}_{\sigma_1}(Q_1) &= \textbf{SELECT} * \textbf{FROM}\ \texttt{student}\ \textbf{WHERE}\ \texttt{id = '0'}, \\
\mathit{SQL}_{\sigma_2}(Q_1) &= \textbf{SELECT} * \textbf{FROM}\ \texttt{student}\ \textbf{WHERE}\ \texttt{id = '5'}.
\end{aligned}$$

Moreover, $\sigma_1(Q_1) = \emptyset$ and $|\sigma_1(Q_1)| = 0$, consistent with the first example execution (Figure 3(a)). Also, $\sigma_2(Q_1)$ contains the row in the `student` table: $\sigma_2(Q_1) = (\{$`student.id` : '5', `student.password` : '2', `student.firstname` : '3', `student.lastname` : '4'$\})$. The row count $|\sigma_2(Q_1)| = 1$ is consistent with the second example execution, where the first query is configured to retrieve at least one row (Figure 5(a)).

Definition 3.8. We denote the *concrete trace* from executing a program $P \in \text{Prog}$ in context $\sigma \in \mathit{Context}$ as $\sigma(P) \in \text{CTrace}$ (Figure 11(a)). A concrete trace consists of the intercepted SQL traffic, specifically, the queries CQuery* and corresponding retrieved rows CData*. Clearly, for any

[3] For brevity, we do not spell out the columns in the `SELECT` clause and the tables in the `WHERE` clause.

$$
\begin{array}{lcl}
\text{CTrace} & := & \text{CQuery}^* \ \text{CData}^* \\
\text{CQuery} & := & \texttt{SELECT} \ \text{CCol}^+ \ \texttt{FROM} \ \text{CJoin} \ \texttt{WHERE} \ \text{CExpr} \\
\text{CJoin} & := & t \mid \text{CJoin} \ \texttt{JOIN} \ t \ \texttt{ON} \ \text{CCol} = \text{CCol} \\
\text{CExpr} & := & \texttt{true} \mid \text{CExpr} \ \texttt{AND} \ \text{CExpr} \mid \text{CCol} = \text{CCol} \mid \text{CCol} = \text{CVal} \mid \text{CCol} \ \texttt{IN} \ \text{CVal}^* \\
\text{CCol} & := & t.c \\
\text{CVal} & := & i \mid s \\
\text{CData} & := & \text{CRow}^* \\
\text{CRow} & := & (\text{CCol} \ \text{CVal})^+
\end{array}
$$

$$t \in \mathit{Table}, \quad c \in \mathit{Column}, \quad i \in \mathit{Int}, \quad s \in \mathit{String}$$

(a) Concrete traces.

$$
\begin{array}{lcl}
\text{ATrace} & := & \text{AQuery}^* \ r^* \\
\text{AQuery} & := & y \leftarrow \texttt{select} \ \text{ACol}^+ \ \texttt{where} \ \text{AExpr} \\
\text{AExpr} & := & \texttt{true} \mid \text{AExpr} \wedge \text{AExpr} \mid \text{ACol} = \text{ACol} \mid \text{ACol} \in \text{AOrig}^+ \\
\text{ACol} & := & t.c \\
\text{AOrig} & := & x \mid y.\text{ACol}
\end{array}
$$

$$x, y \in \mathit{Variable}, \quad t \in \mathit{Table}, \quad c \in \mathit{Column}, \quad r \in \mathbb{Z}_{\geq 0}$$

(b) Abstract traces.

Fig. 11. Grammars for concrete and abstract traces.

query $Q \in$ Query, expression $E \in$ Expr, and context $\sigma \in \mathit{Context}$, we have $SQL_\sigma(Q) \in$ CQuery and $SQL_\sigma(E) \in$ CExpr.

Figure 12 presents the rules for executing a program to obtain a concrete trace. We shall write $[\,^q_d\,]$ for the concrete trace $(q\ d) \in$ CTrace. We write $\cdot @ \cdot$ to denote concatenating two lists into a list.

Remark. In addition to producing a trace of database traffic, the program execution also produces outputs (CVal*) from evaluating Print statements with $\mathit{Print}_\cdot(\cdot)$. As presented, our algorithm (and associated soundness proof) does not work with Print statements. Our implemented Konure prototype infers Print statements by correlating values that appear in the output with values observed in the database traffic. Recall that in the Konure DSL, each Print statement is associated with a query and only prints values retrieved by its query. This restriction enables Konure to associate each Print statement with its corresponding query.

Example 3. Following the notation in Example 1 and Example 2, let Q_2 be the second inferred query in Figure 8, that is,

$$
\begin{aligned}
Q_2 = y_2 \leftarrow\, & \textbf{select}\ \texttt{student.id}, \texttt{student.password}, \texttt{student.firstname}, \texttt{student.lastname} \\
& \textbf{where}\ \texttt{student.id} = s \wedge \texttt{student.password} = p\,;\ \textbf{print}\ \texttt{[]}.
\end{aligned}
$$

Let hypothetical program

$$P' = \texttt{if}\ Q_1\ \texttt{then}\ Q_2\ \texttt{else}\ \epsilon,$$

then executing P' in σ_1 would produce concrete trace:

$$\sigma_1(P') = \begin{bmatrix} \textbf{SELECT} * \textbf{FROM}\ \texttt{student}\ \textbf{WHERE}\ \texttt{id = '0'} \\ \emptyset \end{bmatrix}.$$

$$\frac{}{\sigma(\epsilon) = \begin{bmatrix} \text{Nil} \\ \text{Nil} \end{bmatrix}} \quad \text{(epsilon)}$$

$$\frac{\sigma[Q.y \mapsto \sigma(Q)](P) = \begin{bmatrix} q \\ d \end{bmatrix}}{\sigma(Q\ P) = \begin{bmatrix} SQL_\sigma(Q) @ q \\ \sigma(Q) @ d \end{bmatrix}} \quad \text{(seq)}$$

$$\frac{|\sigma(Q)| > 0 \qquad \sigma[Q.y \mapsto \sigma(Q)](P_1) = \begin{bmatrix} q \\ d \end{bmatrix}}{\sigma(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2) = \begin{bmatrix} SQL_\sigma(Q) @ q \\ \sigma(Q) @ d \end{bmatrix}} \quad \text{(if-1)}$$

$$\frac{|\sigma(Q)| = 0 \qquad \sigma(P_2) = \begin{bmatrix} q \\ d \end{bmatrix}}{\sigma(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2) = \begin{bmatrix} SQL_\sigma(Q) @ q \\ \emptyset @ d \end{bmatrix}} \quad \text{(if-2)}$$

$$\frac{\begin{array}{c} \sigma(Q) = (x_1, \ldots, x_r) \qquad r > 0 \\ \sigma[Q.y \mapsto x_1](P_1) = \begin{bmatrix} q_1 \\ d_1 \end{bmatrix} \qquad \ldots \qquad \sigma[Q.y \mapsto x_r](P_1) = \begin{bmatrix} q_r \\ d_r \end{bmatrix} \end{array}}{\sigma(\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2) = \begin{bmatrix} SQL_\sigma(Q) @ q_1 @ \ldots @ q_r \\ \sigma(Q) @ d_1 @ \ldots @ d_r \end{bmatrix}} \quad \text{(for-1)}$$

$$\frac{|\sigma(Q)| = 0 \qquad \sigma(P_2) = \begin{bmatrix} q \\ d \end{bmatrix}}{\sigma(\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2) = \begin{bmatrix} SQL_\sigma(Q) @ q \\ \emptyset @ d \end{bmatrix}} \quad \text{(for-2)}$$

$$P, P_1, P_2 \in \text{Prog}, \quad Q \in \text{Query}, \quad \sigma \in \mathit{Context}, \quad y \in \mathit{Variable}, \quad r \in \mathbb{Z}_{\geq 0}$$
$$q, q_1, \ldots, q_r \in \text{CQuery}, \quad d, d_1, \ldots, d_r \in \text{CData}, \quad x_1, \ldots, x_r \in \text{CRow}$$

Fig. 12. Semantics for executing a program using a context to obtain a concrete trace.

Executing P' in σ_2 would produce concrete trace:

$$\sigma_2(P') = \begin{bmatrix} \textbf{SELECT} * \textbf{FROM}\ \texttt{student}\ \textbf{WHERE}\ \texttt{id} = \text{'5'}, & \textbf{SELECT} * \textbf{FROM}\ \texttt{student}\ \textbf{WHERE}\ \texttt{id} = \text{'5'} \wedge \texttt{password} = \text{'6'} \\ (\{\texttt{student.id} : \text{'5'}, \texttt{student.password} : \text{'2'}, \texttt{student.firstname} : \text{'3'}, \texttt{student.lastname} : \text{'4'}\}), & \emptyset \end{bmatrix}.$$

These two concrete traces are consistent with the two example traces in Figure 3(a) and Figure 5(a), respectively. So far, the hypothetical program P' is consistent with the observed behavior of the example program (Section 2). However, the third context in the example would cause P' to behave inconsistently (Figure 7).

Definition 3.9. $\boxed{P}$ denotes the black box executable of a program $P \in$ Prog, i.e., executing $\boxed{P}$ in context $\sigma \in \mathit{Context}$ produces the concrete trace $\sigma(P)$. Note that Konure does not access the source code of P when it executes $\boxed{P}$.

Definition 3.10. An *abstract trace* is the list of queries, along with their results, that Konure generates from a concrete trace after replacing concrete values with their origin locations and replacing SQL syntax with the syntax of abstract traces (Figure 11(b)). An abstract trace contains abstract queries (AQuery*) and row counts for each query (r^*). The main modifications from a concrete trace are to replace each concrete value by its origin location and to summarize the retrieved data with row counts.

To infer the origin locations, Konure maintains a context, which keeps track of the concrete values available at each origin location in the input and result components. One complication is the possibility that two distinct origin locations may hold the same concrete value in an execution. When such ambiguities occur, Konure adopts a demand-driven approach to obtain an

$$\text{Tree} \quad := \quad \text{Nil} \mid (Q, r) \searrow \text{Tree} \mid (Q, r) \circlearrowright \text{Tree}^*$$

$$Q \in \text{Query}, \quad r \in \mathbb{Z}_{\geq 0}$$

Fig. 13. Grammar for loop layout trees.

unambiguous origin location (Section 3.4). With the origin locations inferred, it is straightforward to rewrite the trace syntax as an abstract trace.

Example 4. Following the notation in Example 3, the abstract trace for $\sigma_1(P')$ is the same as Figure 3(b), with a row count 0. The abstract trace for $\sigma_2(P')$ is the same as Figure 5(b), with row counts $1, 0$.

Definition 3.11. A *query-result pair* (Q, r) has a query $Q \in \text{Query}$ and an integer $r \in \mathbb{Z}_{\geq 0}$ that counts the number of rows retrieved by Q during execution. Converting an abstract trace into a list of query-result pairs is straightforward.

Example 5. Following the notation in Example 3, the abstract trace for $\sigma_1(P')$ converts into the following list of query-result pairs: $e_1 = (Q_1, 0)$. The abstract trace for $\sigma_2(P')$ converts into the following list of query-result pairs: $e_2 = (Q_1, 1), (Q_2, 0)$.

Definition 3.12. A *loop layout tree* for a program $P \in \text{Prog}$ is a tree that represents information about the execution of loops (Figure 13). Each node in the loop layout tree is a query-result pair that corresponds to a query in P. Each node represents whether a loop in P iterates over the corresponding query multiple times. In particular, when a loop in P iterates over a query multiple times, the query's corresponding node in the loop layout tree has multiple subtrees, with each subtree corresponding to an iteration of the loop. We convert a list of query-result pairs into a loop layout tree in DetectLoops, which we discuss below.

Example 6. Following the notation in Example 3, the loop layout tree for P' executing in σ_1 is:

$$l_1 = (Q_1, 0) \searrow \text{Nil}.$$

The loop layout tree for P' executing in σ_2 is:

$$l_2 = (Q_1, 1) \searrow ((Q_2, 0) \searrow \text{Nil}).$$

Let Q_3, Q_4, Q_5 be the inferred queries for the third, fourth, and fifth queries in Figure 9, respectively. Let hypothetical program

$$P'' = \texttt{if}\ Q_1\ \texttt{then}\ \{\texttt{if}\ Q_2\ \texttt{then}\ \{\texttt{for}\ Q_3\ \texttt{do}\ \{Q_4\ Q_5\}\ \texttt{else}\ \epsilon\}\ \texttt{else}\ \epsilon\}\ \texttt{else}\ \epsilon.$$

Let σ_3 be the context for producing the example trace in Figure 9. When executing P'' in σ_3, the queries Q_1, Q_2, Q_3 retrieve one, one, and two rows, respectively. The loop that iterates over Q_3 is repeated twice. Let r_{41}, r_{51} be the row counts for Q_4, Q_5 in the first iteration of the loop. Let r_{42}, r_{52} be the row counts for Q_4, Q_5 in the second iteration of the loop. The loop layout tree for P'' executing in σ_3 is:

$$\begin{aligned} l_3 = (Q_1, 1) \searrow ((Q_2, 1) \searrow ((Q_3, 2) \circlearrowright ((Q_4, r_{41}) \searrow ((Q_5, r_{51}) \searrow \text{Nil}), \\ (Q_4, r_{42}) \searrow ((Q_5, r_{52}) \searrow \text{Nil})))). \end{aligned}$$

Definition 3.13. An *annotated trace* is an ordered list of annotated query tuples. Each tuple, denoted as $\langle Q, r, \lambda \rangle$, has three components obtained from a query $Q \in \text{Query}$. The first component is the query Q. The second component is the number of rows retrieved by Q during an execution.

The third component is the annotated information of whether a loop was found to iterate over data retrieved by Q. If such loop was found, then λ is a nonnegative integer that indicates the iteration index. If no such loop was found, then $\lambda = \text{NotLoop}$. Each path from the root of the loop layout tree to a leaf generates a corresponding annotated trace.

Example 7. Following the notation in Example 6, executing P' in σ_1 results in an annotated trace:

$$t_1 = \langle Q_1, 0, \text{NotLoop}\rangle.$$

Executing P' in σ_2 results in an annotated trace:

$$t_2 = \langle Q_1, 1, \text{NotLoop}\rangle, \langle Q_2, 0, \text{NotLoop}\rangle.$$

Executing P'' in σ_3 results in two annotated traces:

$$t_{31} = \langle Q_1, 1, \text{NotLoop}\rangle, \langle Q_2, 1, \text{NotLoop}\rangle, \langle Q_3, 2, 1\rangle, \langle Q_4, r_{41}, \text{NotLoop}\rangle, \langle Q_5, r_{51}, \text{NotLoop}\rangle,$$
$$t_{32} = \langle Q_1, 1, \text{NotLoop}\rangle, \langle Q_2, 1, \text{NotLoop}\rangle, \langle Q_3, 2, 2\rangle, \langle Q_4, r_{42}, \text{NotLoop}\rangle, \langle Q_5, r_{52}, \text{NotLoop}\rangle.$$

Definition 3.14. A *path constraint* $W = (\langle Q_1, r_1, s_1\rangle, \ldots, \langle Q_n, r_n, s_n\rangle)$, consists of a sequence of queries $Q_1, \ldots, Q_n \in \text{Query}$, row count constraints $r_1, \ldots, r_n$, and Boolean flags $s_1, \ldots, s_n$. Each r_i specifies the range of the number of rows in a query result, denoted as one of $(= 0)$, (≥ 1), or (≥ 2). Each s_i is `true` if a loop iterates over the corresponding retrieved rows and `false` otherwise.

Example 8. In Section 2, the first execution does not impose any path constraints,

$$W_1 = \text{Nil}.$$

Following the notation in Example 6, the path constraint specifying that Q_1 retrieves at least one row is:

$$W_2 = \langle Q_1, \geq 1, \texttt{false}\rangle.$$

The path constraint specifying that Q_1, Q_2 each retrieves at least one row is:

$$W_3 = (\langle Q_1, \geq 1, \texttt{false}\rangle, \langle Q_2, \geq 1, \texttt{false}\rangle).$$

Before knowing whether a loop iterates over the results of Q_3, the path constraint specifying that Q_1, Q_2, Q_3 retrieve at least one, at least one, and at least two rows, respectively, is:

$$W_4 = (\langle Q_1, \geq 1, \texttt{false}\rangle, \langle Q_2, \geq 1, \texttt{false}\rangle, \langle Q_3, \geq 2, \texttt{false}\rangle).$$

After knowing that a loop iterates over the results of Q_3, the path constraint specifying that Q_1, Q_2, Q_3, Q_4 each retrieves at least one row is:

$$W_5 = (\langle Q_1, \geq 1, \texttt{false}\rangle, \langle Q_2, \geq 1, \texttt{false}\rangle, \langle Q_3, \geq 1, \texttt{true}\rangle, \langle Q_4, \geq 1, \texttt{false}\rangle).$$

Definition 3.15. We define the $\simeq$ operator as follows:

$$r \simeq (= 0) = \begin{cases} \texttt{true} & \text{if } r = 0 \\ \texttt{false} & \text{otherwise} \end{cases}$$
$$r \simeq (\geq 1) = \begin{cases} \texttt{true} & \text{if } r \geq 1 \\ \texttt{false} & \text{otherwise} \end{cases}$$
$$r \simeq (\geq 2) = \begin{cases} \texttt{true} & \text{if } r \geq 2 \\ \texttt{false} & \text{otherwise,} \end{cases}$$

where $r \in \mathbb{Z}_{\geq 0}$.

Definition 3.16. A context $\sigma \in \mathit{Context}$ *satisfies* a path constraint $W = (\langle Q_1, r_1, s_1\rangle, \ldots, \langle Q_n, r_n, s_n\rangle)$ if (1) a sequence of contexts $\sigma_1, \ldots, \sigma_n \in \mathit{Context}$ are updated according to the

$$\frac{}{\texttt{true} \doteq_W \texttt{true}} \quad \text{(true)}$$

$$\frac{}{(C_1 = C_2) \doteq_W (C_1 = C_2)} \quad \text{(col)}$$

$$\frac{O \equiv_W O'}{(C = O) \doteq_W (C = O')} \quad \text{(orig)}$$

$$\frac{E_1 \doteq_W E_1' \qquad E_2 \doteq_W E_2'}{(E_1 \land E_2) \doteq_W \left(E_1' \land E_2'\right)} \quad \text{(and)}$$

$E_1, E_2, E_1', E_2' \in \text{Expr}, \quad C, C_1, C_2 \in \text{Col}, \quad O, O' \in \text{Orig}, \quad W$ is a path constraint

Fig. 14. Check if two expressions are identical except for equivalent variables with respect to a path constraint.

evaluation of the queries $Q_1, \ldots, Q_n$ in σ and (2) $|\sigma_i(Q_i)| \simeq r_i$ for all $i = 1, \ldots, n$. Specifically, the context sequence satisfies $\sigma_1 = \sigma$ and for all $i = 1, \ldots, n-1$:

$$\sigma_{i+1} = \begin{cases} \sigma_i[Q_i.y \mapsto \sigma_i(Q_i)] & \text{if } s_i = \texttt{false} \text{ or } |\sigma_i(Q_i)| = 0 \\ \sigma_i[Q_i.y \mapsto \sigma_i(Q_i)[k_i]] & \text{if } s_i = \texttt{true} \text{ and } |\sigma_i(Q_i)| \geq 1, \end{cases}$$

for some integer k_i such that if $|\sigma_i(Q_i)| \geq 1$, then $1 \leq k_i \leq |\sigma_i(Q_i)|$. We call σ_n the context after updating σ with W.

A context $\sigma \in \mathit{Context}$ always satisfies the trivial path constraint $W = \text{Nil}$.

Example 9. Following the notation in Example 1, Example 6, and Example 8, we have:

(1) σ_1 satisfies W_1 but does not satisfy W_2, W_3, W_4,
(2) σ_2 satisfies W_1, W_2 but does not satisfy W_3, W_4, and
(3) σ_3 satisfies W_1, W_2, W_3, W_4.

These results are consistent with how the example in Section 2 chooses contexts to infer each production.

Definition 3.17. Origin locations $O_1, O_2 \in \text{Orig}$ are equivalent with respect to path constraint W, denoted as $O_1 \equiv_W O_2$, if for any context $\sigma \in \mathit{Context}$ that satisfies W, O_1, O_2 hold the same values in the context after updating σ with W.

Example 10. Following the notation in Example 2, Example 3, and Example 8, we have:

$$s \equiv_{W_2} y_1.\texttt{student.id}, \qquad p \not\equiv_{W_2} y_1.\texttt{student.password},$$
$$s \equiv_{W_3} y_2.\texttt{student.id}, \qquad p \equiv_{W_3} y_2.\texttt{student.password}.$$

Definition 3.18. Expressions $E_1, E_2 \in \text{Expr}$ are identical except for equivalent variables with respect to path constraint W, denoted as $E_1 \doteq_W E_2$, if all of their corresponding origin locations are equivalent with respect to W and all of the remaining components are syntactically identical (Figure 14).

Queries $Q_1, Q_2 \in \text{Query}$ are identical except for equivalent variables with respect to path constraint W and variables Y_1, Y_2, denoted as $Q_1 \doteq_{W, Y_1, Y_2} Q_2$, if the following conditions hold:

(1) $Q_1 = y_1 \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ E_1\ ;\ \texttt{print}\ \overline{O_1}$,
(2) $Q_2 = y_2 \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ E_2\ ;\ \texttt{print}\ \overline{O_2}$, and
(3) $E_1' \doteq_W E_2$, where E_1' is the expression obtained from E_1 after replacing all occurrences of variables in Y_1 with their counterparts in Y_2.

ALGORITHM 1: Infer an executable program

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Output: Program equivalent to P.

procedure INFER($\boxed{P}$)
  $\sigma \leftarrow$ Database empty, input parameters distinct
  $t \leftarrow$ GETTRACE($\boxed{P}$, Nil, σ)
  return INFERPROG($\boxed{P}$, Nil, t)
end procedure

Informally, two queries are identical except for equivalent variables when, after renaming variables and removing Print statements, the queries are syntactically identical except for the use of different but equivalent origin locations.

Example 11. Following the notation in Example 3 and Example 8, let Q_2' be an alternative second inferred query in Figure 8, that is,

$$Q_2' = y_2 \leftarrow \textbf{select}\ \texttt{student.id}, \texttt{student.password}, \texttt{student.firstname}, \texttt{student.lastname}$$
$$\textbf{where}\ \texttt{student.id} = y_1.\texttt{student.id} \wedge \texttt{student.password} = p\,;\ \textbf{print}\ \texttt{[]},$$

then $Q_2 \doteq_{W_2,\mathrm{Nil},\mathrm{Nil}} Q_2'$.

Definition 3.19. An annotated trace $t = \langle Q_1, r_1, \lambda_1\rangle, \ldots, \langle Q_n, r_n, \lambda_n\rangle$ is *consistent with* path constraint W, denoted as $t \sim W$, if the path specified in W is not longer than t, each query in t matches the corresponding query in W, and each row count in t matches the corresponding requirement in W:

$$t \sim \mathrm{Nil} = \texttt{true}$$
$$t \sim (\langle Q_1', r_1', s_1'\rangle, \ldots, \langle Q_m', r_m', s_m'\rangle) = m \le n \wedge \left(\forall i = 1, \ldots, m : r_i \simeq r_i' \wedge Q_i \doteq_{W_i, Y_i, Y_i'} Q_i'\right),$$

where $W_i = (\langle Q_1', r_1', s_1'\rangle, \ldots, \langle Q_{i-1}', r_{i-1}', s_{i-1}'\rangle)$ contains the first $(i-1)$ constraint tuples in W, $Y_i = (Q_1.y, \ldots, Q_{i-1}.y)$ is the list of variables defined by the first $(i-1)$ queries in t, and $Y_i' = (Q_1'.y, \ldots, Q_{i-1}'.y)$ is the list of variables defined by the queries in W_i.

Example 12. Following the notation in Example 7 and Example 8, we have $t_1 \sim W_1$, $t_2 \sim W_1$, $t_{31} \sim W_1$, $t_{32} \sim W_1$, $t_1 \nsim W_2$, $t_2 \sim W_2$, $t_{31} \sim W_2$, $t_{32} \sim W_2$, $t_1 \nsim W_3$, $t_2 \nsim W_3$, $t_{31} \sim W_3$, $t_{32} \sim W_3$, $t_1 \nsim W_4$, $t_2 \nsim W_4$, $t_{31} \sim W_4$, $t_{32} \sim W_4$, $t_1 \nsim W_5$, and $t_2 \nsim W_5$.

If we additionally have, for example, $r_{41} = 0$, $r_{42} = 1$, and $r_{52} = 3$, then $t_{31} \nsim W_5$ and $t_{32} \sim W_5$.

3.2.2 Algorithm. We next present the KONURE inference algorithm, which works with the black box executable of a program (Algorithm 1). KONURE executes the program using carefully chosen contexts that match certain path constraints. Each time the program runs, it produces a concrete trace, from which KONURE constructs an abstract trace and then an annotated trace (Algorithm 2). Conceptually, KONURE follows annotated traces to traverse paths in the program, assuming that the program belongs to the KONURE DSL (Section 3.1). KONURE recursively infers the program by choosing the appropriate production to resolve each nonterminal of the DSL program (Algorithm 6).

INFER: Algorithm 1 takes an executable program $\boxed{P}$. It first configures an initial context σ where all database tables are empty and the input parameters are distinct. It then invokes GETTRACE, which executes $\boxed{P}$ in context σ and returns an initial annotated trace t. Finally, INFER invokes the main KONURE inference algorithm, INFERPROG, to infer P.

ALGORITHM 2: Execute a program and deduplicate the trace according to a path constraint

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Input: W is a path constraint.
Input: σ is a context that satisfies W.
Output: Annotated trace t, $t \sim W$, from executing $\boxed{P}$ with σ.

procedure GetTrace($\boxed{P}$, W, σ)
    $e \leftarrow$ Execute($\boxed{P}$, σ)
    $l \leftarrow$ DetectLoops(e)
    $t \leftarrow$ MatchPath(l, W)
    return t
end procedure

Example 13. In Section 2, Konure invokes Infer to infer the example program. To execute the program for the first time, Konure uses the initial context in variable σ, which equals σ_1 in Example 1. The resulting trace (Figure 3) is converted into an annotated trace, variable t, which equals t_1 in Example 7.

GetTrace: Algorithm 2 takes an executable program $\boxed{P}$, path constraint W, and context σ as parameters. It first invokes Execute, which runs $\boxed{P}$ in context σ to obtain the flat list e of query-result pairs converted from the concrete trace that $\boxed{P}$ generates when it runs. It then invokes DetectLoops, which runs the Konure loop detection algorithm to produce the loop layout tree l. Finally, MatchPath generates an annotated trace that corresponds to a path through l consistent with the path constraint W.

Execute: The Execute procedure takes an executable program $\boxed{P}$ and a context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \mathit{Context}$. It first populates the database with contents specified in σ_D and then executes $\boxed{P}$ with input parameters specified in σ_I. It collects the outputs and database traffic, i.e., the concrete trace $\sigma(P)$ (Figure 1). Execute converts the concrete trace into an abstract trace, converts the abstract trace into a list of query-result pairs, then returns this list of pairs.

Example 14. In Section 2, when Konure executes the program for the first time, it invokes Execute with context σ_1 (Example 1). Execute configures the database to empty and runs the program with inputs `'0'` and `'1'`. This execution results in the first concrete trace (Figure 3(a)), which equals $\sigma_1(P')$ in Example 3. Execute converts the concrete trace into an abstract trace, described in Example 4, and then into a list of query-result pairs that equals e_1 in Example 5.

DetectLoops: Algorithm 3 takes a list of query-result pairs and constructs a loop layout tree. (1) If the first query retrieves $r \geq 2$ rows, DetectLoops checks if the skeleton of the second query is repeated exactly r times in the tail of the trace. If the repetitions match, DetectLoops determines that a loop iterates over the first query in the trace, splits the trace into r segments that each correspond to an iteration of the loop, recursively constructs a loop layout tree for each segment, and then inserts the recursively constructed loop layout trees as the children of the first query. (2) In all other scenarios, DetectLoops determines that no loop iterates over the first query in the trace, recursively constructs a loop layout tree for the tail of the trace, and then inserts the recursively constructed loop layout tree as the child of the first query of the trace.

Example 15. Following the notation in Example 5 and Example 6, we have DetectLoops$(e_1) = l_1$ and DetectLoops$(e_2) = l_2$. Let e_3 be the list of query-result pairs resulting from executing P'' in σ_3 (Example 6), that is, $e_3 = (Q_1, 1), (Q_2, 1), (Q_3, 2), (Q_4, r_{41}), (Q_5, r_{51}), (Q_4, r_{42}), (Q_5, r_{52})$, then DetectLoops$(e_3) = l_3$.

ALGORITHM 3: Loop detection algorithm

Input: e is either Nil or a nonempty list of query-result pairs $(Q_1, r_1), \ldots, (Q_n, r_n)$.
Output: Loop layout tree constructed from e.

procedure DetectLoops(e)
    if e = Nil **then**
        return Nil
    end if
    $(Q_1, r_1), \ldots, (Q_n, r_n) \leftarrow e$
    $a \leftarrow$ empty list
    for $j \leftarrow 2, 3, \ldots, n$ **do** ▷ Identify repetitions
        if $\pi_{\mathbb{S}} Q_j = \pi_{\mathbb{S}} Q_2$ **then**
            Append j to a
        end if
    end for
    if $r_1 \leq 1$ **or** $r_1 \neq \texttt{len}(a)$ **then** ▷ Did not find repetitions caused by any loops that iterate over Q_1
        $e' \leftarrow (Q_2, r_2), \ldots, (Q_n, r_n)$
        $l \leftarrow$ DetectLoops(e')
        return $(Q_1, r_1) \searrow l$
    else ▷ Found a loop that iterates over Q_1
        Append $n + 1$ to a
        for $j \leftarrow 1, 2, \ldots, r_1$ **do**
            $b \leftarrow a[j]$
            $c \leftarrow a[j + 1] - 1$
            $e' \leftarrow (Q_b, r_b), \ldots, (Q_c, r_c)$
            $l_j \leftarrow$ DetectLoops(e')
        end for
        return $(Q_1, r_1) \circlearrowleft (l_1, \ldots, l_{r_1})$
    end if
end procedure

MatchPath: Algorithm 4 takes a loop layout tree and a path constraint. The procedure first calls GetAnnotatedTrace to convert the loop layout tree into a set of annotated traces that each contains at most one iteration of any loop. MatchPath then picks an annotated trace that is consistent with the given path constraint.

Example 16. Following the notation in Example 6 and Example 7, we have GetAnnotatedTrace(l_1) = $\{t_1\}$, GetAnnotatedTrace(l_2) = $\{t_2\}$, and GetAnnotatedTrace(l_3) = $\{t_{31}, t_{32}\}$. Note that the annotated traces t_{31} and t_{32} each contains only one iteration of the loop, even though this loop is repeated multiple times.

Following the notation in Example 8, we have MatchPath(l_1, W_1) = t_1, MatchPath(l_2, W_2) = t_2, MatchPath(l_3, W_3) = t_{31} (or t_{32}, depending on the order in which MatchPath enumerates the traces returned from GetAnnotatedTrace), and MatchPath(l_3, W_4) = t_{31} (or t_{32}).

If we additionally have, for example, $r_{41} = 0$, $r_{42} = 1$, and $r_{52} = 3$, then MatchPath(l_3, W_5) = t_{32}. In this case t_{31} can no longer be returned, because $t_{31} \nsim W_5$ (Example 12).

MakePathConstraint: The MakePathConstraint procedure takes an annotated trace prefix t, a subsequent query $Q \in$ Query, and an integer $r \in \mathbb{Z}_{\geq 0}$. The procedure constructs a new path constraint, W, which specifies that any satisfying context must enable the program to execute down the same path as t, then perform query Q and retrieve a certain number of rows as specified by r. In particular, if $r = 0$, then Q is required to retrieve zero rows. If $r = 1$ or $r = 2$, then Q is required to retrieve at least r rows. More concretely, for each annotated query tuple

ALGORITHM 4: Pick an annotated trace that is consistent with a path constraint

Input: l is a loop layout tree.
Input: W is a path constraint.
Output: Annotated trace constructed from l that is consistent with W.

procedure MatchPath(l, W)
  for t **in** GetAnnotatedTrace(l) **do**
    if t = Nil **then**
      continue
    end if
    if $t \sim W$ **then**
      return t
    end if
  end for
  return Nil
end procedure

Input: l is a loop layout tree.
Output: Set of annotated traces constructed from l.

procedure GetAnnotatedTrace(l)
  if l = Nil **then**
    return { Nil }
  else if $l = (Q, r) \searrow l'$ **then**
    return $\{ \langle Q, r, \text{NotLoop} \rangle \,@\, t \mid t \in$ GetAnnotatedTrace$(l') \}$
  else if $l = (Q, r) \circlearrowright \left(l'_1, l'_2, \ldots, l'_r\right)$ **then** ▷ $r \geq 2$
    return $\cup_{i=1}^{r} \{ \langle Q, r, i \rangle \,@\, t \mid t \in$ GetAnnotatedTrace$(l'_i) \}$
  end if
end procedure

ALGORITHM 5: Obtain a deduplicated annotated trace that satisfies a path constraint

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Input: W is a path constraint.
Output: The first component represents the satisfiability of W. When satisfiable, the second component is an annotated trace t where $t \sim W$.

procedure SolveAndGetTrace($\boxed{P}, W$)
  $\sigma \leftarrow$ Solve(W)
  if σ = Unsat **then**
    return `false`, Nil
  else
    $t \leftarrow$ GetTrace($\boxed{P}, W, \sigma$)
    return `true`, t
  end if
end procedure

$\langle Q_i, r_i, \lambda_i \rangle$ in t, the procedure adds $\langle Q_i, r'_i, s'_i \rangle$ to the path constraint where $r'_i = \begin{cases} (=0) & \text{if } r_i = 0 \\ (\geq 1) & \text{if } r_i \geq 1 \end{cases}$ and s'_i = `true` if and only if previous recursions of InferProg chose the production "Prog := For" for the corresponding query. The procedure then adds $\langle Q, r', \texttt{false} \rangle$ to the path constraint where $r' = \begin{cases} (=0) & \text{if } r = 0 \\ (\geq 1) & \text{if } r = 1 \\ (\geq 2) & \text{if } r = 2 \end{cases}$.

ALGORITHM 6: Recursively infer a subprogram

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Input: s_1 is a prefix of an annotated trace.
Input: s_2 is a suffix of an annotated trace.
Output: Subprogram equivalent to P's subprogram after trace s_1.

```
procedure InferProg([P], s1, s2)
    if s2 = Nil then return ϵ                                          ▷ Prog := ϵ
    end if
    k ← The length of s1
    Q ← The first query in s2
    for i = 0, 1, 2 do
        W_i ← MakePathConstraint(s1, Q, i)
        (f_i, t_i) ← SolveAndGetTrace([P], W_i)
        if f_i then                                                     ▷ Satisfiable
            t_{i,1} ← t_i[1, ..., (k + 1)]                             ▷ New trace prefix
            t_{i,2} ← t_i[(k + 2), ...]                                ▷ New trace suffix
        end if
    end for
    if f_2 and found loop on the last query in t_{2,1} then
        b_t ← InferProg([P], t_{2,1}, t_{2,2})
        if f_0 then b_f ← InferProg([P], t_{0,1}, t_{0,2})
        else b_f ← ϵ
        end if
        return "for Q do b_t else b_f"                                  ▷ Prog := For
    else if f_0 and f_1 and ((t_{0,2} = Nil and t_{1,2} ≠ Nil) or (t_{0,2} ≠ Nil and t_{1,2} = Nil) or
                the first queries in t_{0,2} and t_{1,2} have different skeletons) then
        b_t ← InferProg([P], t_{1,1}, t_{1,2})
        b_f ← InferProg([P], t_{0,1}, t_{0,2})
        return "if Q then b_t else b_f"                                 ▷ Prog := If
    else
        if f_0 then b ← InferProg([P], t_{0,1}, t_{0,2})
        else b ← InferProg([P], t_{1,1}, t_{1,2})
        end if
        return "Q b"                                                    ▷ Prog := Seq
    end if
end procedure
```

Example 17. Following the notation in Example 6 and Example 8, we have:

$$\textsc{MakePathConstraint}(\text{Nil}, Q_1, 1) = W_2,$$
$$\textsc{MakePathConstraint}(\langle Q_1, 1, \text{NotLoop}\rangle, Q_2, 1) = W_3,$$
$$\textsc{MakePathConstraint}((\langle Q_1, 1, \text{NotLoop}\rangle, \langle Q_2, 1, \text{NotLoop}\rangle), Q_3, 2) = W_4.$$

InferProg: Algorithm 6 implements the main Konure inference algorithm. This algorithm recursively explores all relevant paths through the program, resolving Prog nonterminals as they are (conceptually) encountered. Algorithm 6 takes as parameters the executable $\boxed{P}$ of the program to infer and a split annotated trace consisting of a prefix s_1 that corresponds to an explored path through the program and a suffix s_2 from the remaining unexplored part of the program. The first Query Q in s_2 is generated by the next Prog nonterminal to resolve. Konure therefore determines whether the query Q was generated by a Seq, If, or For statement, then recurses to infer the remaining parts of the program.

Konure makes this determination by examining three deduplicated annotated traces t_0, t_1, and t_2. All of these traces are from executions that follow the same path to Q as s_1. In the execution that generated t_0, Q retrieves zero rows, in the execution that generated t_1, Q retrieves at least one row, and in the execution that generated t_2, Q retrieves at least two rows. If Konure detects a loop in t_2 over the rows that Q retrieves, it infers that Q was generated by a For statement (line 14 in Algorithm 6). Otherwise, it examines t_0 and t_1 to determine if Q was generated by an If statement (line 20 in Algorithm 6) or a Seq statement (line 24 in Algorithm 6)—conceptually, if the queries that follow Q in t_0 and t_1 differ, then Q is generated by an If statement, otherwise it is generated by a Seq statement.

Konure obtains traces t_0, t_1, and t_2 by using MakePathConstraint to construct three path constraints W_0, W_1, and W_2, then using an SMT solver to obtain contexts σ_0, σ_1, and σ_2 that cause $\boxed{P}$ to produce (deduplicated annotated) traces t_0, t_1, and t_2 (Algorithm 5). If W_i is satisfiable, then $t_i \sim W_i$.

Example 18. Consider the first execution of the example program in Section 2. Infer invokes GetTrace with context σ_1 (Example 13). The initial path constraint is W_1 = Nil (Example 8). GetTrace invokes Execute with σ_1, resulting in the list of query-result pairs e_1 (Example 14). Recall from Example 15 that $\text{DetectLoops}(e_1) = l_1$. Recall from Example 16 that $\text{MatchPath}(l_1, W_1) = t_1$. Hence, t_1 is the initial annotated trace obtained from GetTrace.

Infer then invokes InferProg with trace prefix Nil and trace suffix t_1. The first query in t_1 is Q_1 (Example 7). InferProg invokes MakePathConstraint three times, constructing three different path constraints. The first path constraint specifies that Q_1 retrieves zero rows:

$$\text{MakePathConstraint}(\text{Nil}, Q_1, 0) = \langle Q_1, = 0, \texttt{false} \rangle.$$

The second path constraint specifies that Q_1 retrieves at least one row:

$$\text{MakePathConstraint}(\text{Nil}, Q_1, 1) = \langle Q_1, \geq 1, \texttt{false} \rangle.$$

The third path constraint specifies that Q_1 retrieves at least two rows:

$$\text{MakePathConstraint}(\text{Nil}, Q_1, 2) = \langle Q_1, \geq 2, \texttt{false} \rangle.$$

InferProg then invokes SolveAndGetTrace to determine if these path constraints are satisfiable and, if so, obtain the corresponding annotated traces. In the example (Section 2), the first path constraint results in the annotated trace t_1. The second path constraint results in the annotated trace t_2 (Example 7). The third path constraint is not satisfiable. Based on these results, InferProg applies the "Prog := If" production to resolve the topmost Prog nonterminal.

Intuition: InferProg implements the core recursion of the Konure inference algorithm. For any program $P \in \mathcal{K}$, each Prog nonterminal in the abstract syntax tree of P corresponds to a recursive call to InferProg as follows: Each step of the recursion resolves a Prog nonterminal by applying the appropriate production, that is, one of Prog := ϵ, Prog := Seq, Prog := If, and Prog := For (Figure 4). The appropriate production is the (only) one that is consistent with the incoming trace, $s_1 @ s_2$, as well as three other potential traces, t_0, t_1, and t_2. InferProg recurses only after collecting sufficient information to uniquely determine the correct production for the current Prog nonterminal. As a result, this recursion does not need to backtrack.

Note that all of the traces used in InferProg are deduplicated annotated traces. Because each annotated trace corresponds to a path through the program AST, the length of the annotated trace is bounded by the code size of P. Because each recursive call to InferProg consumes a tuple in the incoming trace ($s_1 @ s_2$), the number of recursive calls to InferProg is bounded by the maximum length of annotated traces, which is bounded by the size of P. Although $\mathcal{K}$ can express arbitrarily large programs, each program has a finite code size. Hence, Infer($\boxed{P}$) terminates for any program $P \in \mathcal{K}$. We present these properties in Section 4.

3.3 Path Constraint Solver

Solve takes a path constraint W and uses an SMT solver to solve for a context $\sigma \in \mathit{Context}$ that satisfies W. The procedure returns a satisfying σ if it exists and returns "Unsat" otherwise.

Like many database test data generation approaches [26, 44, 75, 77, 78, 81], Solve uses a row-based approach to translate path constraints into SMT formulas. For each query Q_i in W that is required to retrieve at least one or at least two rows, Solve generates variables that model the required number of rows of the relevant tables. It then generates constraints that require the values of these variables to satisfy the selection criteria of Q_i. It also generates constraints that require primary keys to be unique.

For each query Q_i that is required to retrieve zero rows, Solve generates constraints that ensure that none of the values in the relevant tables satisfy the selection criteria of Q_i. If Q_i occurs in a loop, the constraints only enforce that Q_i retrieves zero rows in at least one iteration of the loop (as opposed to always retrieving zero rows). Here, loop iterations map easily to the rows of unknown variables, because loops in the Konure DSL are designed to iterate over rows of data.

3.4 Origin Location Disambiguation

Recall that an origin location $O \in \text{Orig}$ in a program $P \in \text{Prog}$ is an occurrence of a variable x or a column reference y.Col in P. Concrete traces contain intercepted queries executed by the program. In these intercepted queries, the origin locations have been replaced by the corresponding concrete values from the execution. When Konure converts concrete traces into abstract traces, it restores the origin locations by matching concrete values across query results and input parameters to translate the concrete values back into their corresponding origin locations.

Because Konure uses a general SMT solver to obtain contexts σ that satisfy specified path constraints W, the contexts may introduce ambiguity by coincidentally generating the same value in different input parameters or database locations. This ambiguity shows up as different origin locations O_1 and O_2 that both contain the same concrete value to translate. Konure resolves the ambiguity as follows:

- Konure first asks the solver if it is possible to reproduce the path to the ambiguous concrete value with the additional constraint that O_1 and O_2 hold disjoint values. If so, the resulting execution resolves the ambiguity.
- Otherwise, Konure asks the solver if it is possible to reproduce this path with the additional constraint that O_1 holds a value not in O_2. If not, the values in O_1 are a subset of the values in O_2. Konure similarly uses the solver to determine if the values in O_2 are a subset of the values in O_1. If O_1 and O_2 are subsets of each other, they hold the same values and Konure can use either origin location.
- Otherwise, there exists an execution in which O_1 has at least one value v not in O_2 (or vice versa). Konure asks the solver to produce a context that generates this execution. The resulting execution in this context resolves the ambiguity—if the value v ever appears in the same location as the concrete value, then Konure uses O_1 as the origin location, otherwise it uses O_2.

$$\frac{P \doteq_{\mathrm{Nil},\mathrm{Nil},\mathrm{Nil}} P'}{P \doteq P'} \quad \text{(full)}$$

$$\frac{}{\epsilon \doteq_{W,Y,Y'} \epsilon} \quad \text{(epsilon)}$$

$$\frac{Q \doteq_{W,Y,Y'} Q' \qquad Y_Q = Y \mathbin{@} Q.y \qquad Y'_Q = Y' \mathbin{@} Q'.y \qquad P \doteq_{W,Y_Q,Y'_Q} P'}{Q\ P \doteq_{W,Y,Y'} Q'\ P'} \quad \text{(seq)}$$

$$\frac{\begin{array}{c} W_0 = W \mathbin{@} \langle Q', = 0, \texttt{false}\rangle \qquad W_1 = W \mathbin{@} \langle Q', \geq 1, \texttt{false}\rangle \qquad Y_Q = Y \mathbin{@} Q.y \qquad Y'_Q = Y' \mathbin{@} Q'.y \\ Q \doteq_{W,Y,Y'} Q' \qquad P_1 \doteq_{W_1,Y_Q,Y'_Q} P'_1 \qquad P_2 \doteq_{W_0,Y_Q,Y'_Q} P'_2 \end{array}}{\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \doteq_{W,Y,Y'} \texttt{if}\ Q'\ \texttt{then}\ P'_1\ \texttt{else}\ P'_2} \quad \text{(if)}$$

$$\frac{\begin{array}{c} W_0 = W \mathbin{@} \langle Q', = 0, \texttt{true}\rangle \qquad W_1 = W \mathbin{@} \langle Q', \geq 1, \texttt{true}\rangle \qquad Y_Q = Y \mathbin{@} Q.y \qquad Y'_Q = Y' \mathbin{@} Q'.y \\ Q \doteq_{W,Y,Y'} Q' \qquad P_1 \doteq_{W_1,Y_Q,Y'_Q} P'_1 \qquad P_2 \doteq_{W_0,Y_Q,Y'_Q} P'_2 \end{array}}{\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2 \doteq_{W,Y,Y'} \texttt{for}\ Q'\ \texttt{do}\ P'_1\ \texttt{else}\ P'_2} \quad \text{(for)}$$

$$P, P_1, P_2, P', P'_1, P'_2 \in \text{Prog}, \quad Q, Q' \in \text{Query}, \quad y \in \mathit{Variable},$$
$$W, W_0, W_1 \text{ are path constraints}, \quad Y, Y_Q, Y', Y'_Q \in \overline{\mathit{Variable}}$$

Fig. 15. Check if two programs are identical except for equivalent variables.

4 SOUNDNESS PROOF

In this section, we first outline the structure of a soundness proof for the core Konure inference algorithm (Algorithm 1) and then provide the full proof. The proof is structured as follows: Sections 4.1 and 4.2 elaborate on the transformation and the functions that are used to define $\mathcal{K}$ in Section 3.1 (Definition 3.4). Section 4.3 proves Theorem 1. Section 4.4 proves Theorem 2. Section 4.5 proves Theorem 3 and Theorem 4. Section 4.6 proves Theorem 5.

Definition 4.1. Programs $P_1, P_2 \in$ Prog are *identical except for equivalent variables*, denoted as $P_1 \doteq P_2$, if they have the same control structures and if all of the corresponding queries are identical except for equivalent variables with respect to the paths that reach these queries (Figure 15).

Informally, two programs are identical except for equivalent variables when, after renaming variables and removing Print statements, the programs are syntactically identical except for the use of different but equivalent origin locations.

To simplify the presentation, when the context is clear, we write $P_1 \doteq P_2$ as a shorthand for $P_1 \doteq_{W,Y_1,Y_2} P_2$ and write $Q_1 \doteq Q_2$ as a shorthand for $Q_1 \doteq_{W,Y_1,Y_2} Q_2$. By default, we keep track of W, Y_1, Y_2 by traversing the program in the same manner as in Figure 15.

Definition 4.2. For a program $P \in$ Prog and a context $\sigma \in \mathit{Context}$, $\sigma \vdash P \Downarrow_{\text{exec}} e$ denotes evaluating P in σ to obtain a list of query-result pairs e. Figure 16 defines this evaluation.

$\sigma \vdash P \Downarrow_{\text{loops}} l$ denotes evaluating P in σ to obtain a loop layout tree l. Figure 17 defines this evaluation.

Definition 4.3. For programs $P, P' \in$ Prog and annotated trace t, we use the notation $P \xrightarrow{t} P'$ to denote that traversing the AST of P from top to bottom, by following the row counts in t, leads to a subtree P'. Figure 18 defines this traversal.

$$\frac{}{\sigma \vdash \epsilon \Downarrow_{\text{exec}} \text{Nil}} \qquad \text{(epsilon)}$$

$$\frac{\sigma[Q.y \mapsto \sigma(Q)] \vdash P \Downarrow_{\text{exec}} e}{\sigma \vdash Q\ P \Downarrow_{\text{exec}} (Q, |\sigma(Q)|) \mathbin{@} e} \qquad \text{(seq)}$$

$$\frac{|\sigma(Q)| > 0 \qquad \sigma[Q.y \mapsto \sigma(Q)] \vdash P_1 \Downarrow_{\text{exec}} e}{\sigma \vdash \texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2 \Downarrow_{\text{exec}} (Q, |\sigma(Q)|) \mathbin{@} e} \qquad \text{(if-1)}$$

$$\frac{|\sigma(Q)| = 0 \qquad \sigma \vdash P_2 \Downarrow_{\text{exec}} e}{\sigma \vdash \texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2 \Downarrow_{\text{exec}} (Q, 0) \mathbin{@} e} \qquad \text{(if-2)}$$

$$\frac{\begin{array}{c}\sigma(Q) = (x_1, \ldots, x_r) \qquad r > 0 \\ \sigma[Q.y \mapsto x_1] \vdash P_1 \Downarrow_{\text{exec}} e_1 \qquad \ldots \qquad \sigma[Q.y \mapsto x_r] \vdash P_1 \Downarrow_{\text{exec}} e_r\end{array}}{\sigma \vdash \texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2 \Downarrow_{\text{exec}} (Q, r) \mathbin{@} e_1 \mathbin{@} \ldots \mathbin{@} e_r} \qquad \text{(for-1)}$$

$$\frac{|\sigma(Q)| = 0 \qquad \sigma \vdash P_2 \Downarrow_{\text{exec}} e}{\sigma \vdash \texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2 \Downarrow_{\text{exec}} (Q, 0) \mathbin{@} e} \qquad \text{(for-2)}$$

$$P, P_1, P_2 \in \text{Prog}, \quad Q \in \text{Query}, \quad \sigma \in \mathit{Context}, \quad y \in \mathit{Variable}, \quad r \in \mathbb{Z}_{\geq 0}, \quad x_1, \ldots, x_r \in \text{CRow}$$

Fig. 16. Semantics for executing a program using a context to directly obtain a list of query-result pairs.

$$\frac{}{\sigma \vdash \epsilon \Downarrow_{\text{loops}} \text{Nil}} \qquad \text{(epsilon)}$$

$$\frac{\sigma[Q.y \mapsto \sigma(Q)] \vdash P \Downarrow_{\text{loops}} l}{\sigma \vdash Q\ P \Downarrow_{\text{loops}} (Q, |\sigma(Q)|) \searrow l} \qquad \text{(seq)}$$

$$\frac{|\sigma(Q)| > 0 \qquad \sigma[Q.y \mapsto \sigma(Q)] \vdash P_1 \Downarrow_{\text{loops}} l}{\sigma \vdash \texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2 \Downarrow_{\text{loops}} (Q, |\sigma(Q)|) \searrow l} \qquad \text{(if-1)}$$

$$\frac{|\sigma(Q)| = 0 \qquad \sigma \vdash P_2 \Downarrow_{\text{loops}} l}{\sigma \vdash \texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2 \Downarrow_{\text{loops}} (Q, 0) \searrow l} \qquad \text{(if-2)}$$

$$\frac{\begin{array}{c}\sigma(Q) = (x_1, \ldots, x_r) \qquad r \geq 2 \\ \sigma[Q.y \mapsto x_1] \vdash P_1 \Downarrow_{\text{loops}} l_1 \qquad \ldots \qquad \sigma[Q.y \mapsto x_r] \vdash P_1 \Downarrow_{\text{loops}} l_r\end{array}}{\sigma \vdash \texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2 \Downarrow_{\text{loops}} (Q, r) \circlearrowleft (l_1, \ldots, l_r)} \qquad \text{(for-1a)}$$

$$\frac{|\sigma(Q)| = 1 \qquad \sigma[Q.y \mapsto \sigma(Q)] \vdash P_1 \Downarrow_{\text{loops}} l}{\sigma \vdash \texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2 \Downarrow_{\text{loops}} (Q, 1) \searrow l} \qquad \text{(for-1b)}$$

$$\frac{|\sigma(Q)| = 0 \qquad \sigma \vdash P_2 \Downarrow_{\text{loops}} l}{\sigma \vdash \texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2 \Downarrow_{\text{loops}} (Q, 0) \searrow l} \qquad \text{(for-2)}$$

$$P, P_1, P_2 \in \text{Prog}, \quad Q \in \text{Query}, \quad \sigma \in \mathit{Context}, \quad y \in \mathit{Variable}, \quad r \in \mathbb{Z}_{\geq 0}, \quad x_1, \ldots, x_r \in \text{CRow}$$

Fig. 17. Semantics for executing a program using a context to obtain a loop layout tree.

$$\frac{}{P \xrightarrow{\text{Nil}} P} \quad \text{(nil)}$$

$$\frac{P \xrightarrow{t} P' \qquad Q \doteq Q'}{Q\ P \xrightarrow{\langle Q', r, \lambda \rangle \,@\, t} P'} \quad \text{(seq)}$$

$$\frac{r > 0 \qquad P_1 \xrightarrow{t} P_1' \qquad Q \doteq Q'}{\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\langle Q', r, \lambda \rangle \,@\, t} P_1'} \quad \text{(if-1)}$$

$$\frac{P_2 \xrightarrow{t} P_2' \qquad Q \doteq Q'}{\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\langle Q', 0, \lambda \rangle \,@\, t} P_2'} \quad \text{(if-2)}$$

$$\frac{r > 0 \qquad P_1 \xrightarrow{t} P_1' \qquad Q \doteq Q'}{\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\langle Q', r, \lambda \rangle \,@\, t} P_1'} \quad \text{(for-1)}$$

$$\frac{P_2 \xrightarrow{t} P_2' \qquad Q \doteq Q'}{\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2 \xrightarrow{\langle Q', 0, \lambda \rangle \,@\, t} P_2'} \quad \text{(for-2)}$$

$$P, P_1, P_2, P', P_1', P_2' \in \text{Prog}, \quad Q, Q' \in \text{Query}, \quad r \in \mathbb{Z}_{\geq 0}, \quad \lambda \in \mathbb{Z}_{\geq 0} \cup \{\text{NotLoop}\}$$

Fig. 18. Traverse a program by following an annotated trace to obtain a subprogram

$$\frac{}{l \xhookrightarrow{\text{Nil}} l} \quad \text{(nil)}$$

$$\frac{l \xhookrightarrow{t'} l' \qquad Q \doteq Q'}{(Q, r) \searrow l \xhookrightarrow{\langle Q', r, \text{NotLoop} \rangle \,@\, t'} l'} \quad \text{(next)}$$

$$\frac{1 \leq i \leq r \qquad l_i \xhookrightarrow{t'} l' \qquad Q \doteq Q'}{(Q, r) \circlearrowleft (l_1, \ldots, l_r) \xhookrightarrow{\langle Q', r, i \rangle \,@\, t'} l'} \quad \text{(iter)}$$

$$Q, Q' \in \text{Query}, \quad r, i \in \mathbb{Z}_{\geq 0}$$

Fig. 19. Traverse a loop layout tree by following an annotated trace to obtain a subtree

For loop layout trees l, l' and annotated trace t, we use the notation $l \xhookrightarrow{t} l'$ to denote that traversing l from top to bottom, by following the row counts and loop iteration numbers in t, leads to a subtree l'. Figure 19 defines this traversal.

Definition 4.4. A path constraint W is *derived from* a program $P \in \text{Prog}$ if one of the following holds:

(1) $W = \text{Nil}$.
(2) $W = \langle Q', r', s' \rangle$, $P \neq \epsilon$, and $Q' \doteq_{\text{Nil},\text{Nil},\text{Nil}} \mathcal{F}(P)$, where $\mathcal{F}(P)$ is the first query in P. We formally define the function $\mathcal{F}(\cdot)$ in Section 4.2.

(3) $W = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$, where $m \geq 2$, and there exists an annotated trace t such that:
   (a) $P \xrightarrow{t} \epsilon$,
   (b) $t \sim W'$, where $W' = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_{m-1}, r'_{m-1}, s'_{m-1} \rangle, \langle Q'_m, r', s'_m \rangle)$ for some row constraint r', and
   (c) for all $i = 1, \ldots, m-1$, $s'_i = \texttt{true}$ if and only if the corresponding element in P is a `for`-construct.

Definition 4.5. The *size* of a program $P \in \text{Prog}$ is denoted as $\|P\|$ and defined as the number of times that the AST of P applies a production to expand a "Prog" nonterminal:

$$\begin{aligned} \|\epsilon\| &= 1 \\ \|Q\ P\| &= 1 + \|P\| \\ \|\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2\| &= 1 + \|P_1\| + \|P_2\| \\ \|\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2\| &= 1 + \|P_1\| + \|P_2\|, \end{aligned}$$

where $P, P_1, P_2 \in \text{Prog}$ and $Q \in \text{Query}$.

Proposition 4.6 (Solver). *For any path constraint W, the procedure* Solve(W) *returns a context $\sigma \in$ Context if and only if W is satisfiable.*

Rationale. The path constraint solver outlined in Section 3.3 asks the SMT solver a question that is equisatisfiable as the existence of a satisfying context. Since the logical formulas are quantifier-free and involve only equality checks, their satisfiability is efficiently decidable [20]. □

Proposition 4.7 (Disambiguation). *For any program $P \in \mathcal{K}$ and context $\sigma \in$ Context, if $\sigma \vdash P \Downarrow_{exec} e$,* Execute$(\boxed{P}, \sigma) = e'$*, and $e = ((Q_1, r_1), \ldots, (Q_n, r_n))$, then $e' = ((Q'_1, r_1), \ldots, (Q'_n, r_n))$, where $Q_i \doteq Q'_i$ for any $i = 1, \ldots, n$.*

Rationale. The disambiguation procedure (Section 3.4) asks the SMT solver a question that equivalently encodes the relationship between origin locations. By Proposition 4.6, we obtain a correct list of query-result pairs after disambiguating the traces obtained from program execution. □

This proposition states that, after Konure executes the program as a black box and obtains an abstract trace, the resulting list of query-result pairs is equivalent to the outcome from evaluating the source code as in Figure 16.

Theorem 1 (Loop Detection). *For any program $P \in \mathcal{K}$ and context $\sigma \in$ Context, if $\sigma \vdash P \Downarrow_{exec} e$ and $\sigma \vdash P \Downarrow_{loops} l$, then* DetectLoops$(e) = l$.

Proof Sketch. By induction on the derivation of P. We present a full proof in Section 4.3. □

This theorem states that DetectLoops correctly identifies repetitions in the trace caused by loops in a program in $\mathcal{K}$. In particular, the algorithm produces a loop layout tree, same as the outcome of extracting loop information from the program's source code.

Theorem 2 (Trace-Code Correspondence). *For any program $P \in \mathcal{K}$, path constraint W that is derived from P, context $\sigma \in$ Context that satisfies W, and annotated trace t, if $t =$* GetTrace$(\boxed{P}, W, \sigma)$*, then there exists a loop layout tree l' such that:*

(1) $\sigma \vdash P \Downarrow_{loops} l'$,
(2) $t \sim W$,
(3) $P \xrightarrow{t} \epsilon$,

(4) $l' \overset{t}{\hookrightarrow} Nil$*, and*
(5) l' *and the variable* l *are identical except for equivalent variables.*

Proof Sketch. By induction on the derivation of P. We present a full proof in Section 4.4. □

This theorem states that GetTrace correctly extracts from the program execution an annotated trace that corresponds to a path through the program AST. This property ensures that the length of the annotated trace is bounded by the code size of the program in $\mathcal{K}$. This annotated trace also corresponds to a path through the loop layout tree, which enables the core inference algorithm to identify the location of loops in the trace. This annotated trace also satisfies the given path constraint. This property is nontrivial, because when the program executes multiple iterations of a loop, not all of the iterations are required to satisfy the path constraint.

Theorem 3 (Core Recursion). *For any programs* $P \in Prog$ *and* $P' \in \mathcal{K}$ *and annotated traces* t, t'*, if* $P' \xrightarrow{t'} P$ *and* $P \xrightarrow{t} \epsilon$*, then* $P \doteq \textsc{InferProg}(\boxed{P'}, t', t)$*.*

Proof Sketch. The proof first performs case analysis of the relationship between the possible first production in P, properties of the path constraints W_i, and values $f_i, t_{i,j}$ from the executions of $\boxed{P'}$ in Algorithm 6 to show that Algorithm 6 chooses the correct first production in P. The proof then proceeds by induction on the productions applied to derive P. We present a full proof in Section 4.5. □

This theorem states that each recursive call to InferProg correctly returns a subprogram of the final AST.

Theorem 4 (Soundness of Inference). *For any program* $P \in \mathcal{K}$, $P \doteq \textsc{Infer}(\boxed{P})$*.*

Proof Sketch. The proof first shows that the initial trace t at line 3 of Algorithm 1 satisfies $P \xrightarrow{t} \epsilon$. The rest of the proof follows from Theorem 3. We present a full proof in Section 4.5. □

This theorem states our main soundness claim: If a program belongs to $\mathcal{K}$, then Konure infers the correct program.

Theorem 5 (Complexity). *For any program* $P \in \mathcal{K}$*, the execution of* $\textsc{Infer}(\boxed{P})$ *calls the* InferProg *procedure at most* $||P||$ *times.*

Proof Sketch. By induction on the derivation of P. We present a full proof in Section 4.6. □

Intuition: Each recursive call to InferProg constructs a subprogram for $P \in \mathcal{K}$. The algorithm does not need to backtrack, because it never makes an incorrect hypothesis choice. Each step is conclusive—only one nonterminal expansion is possible. The algorithm also does not involve an equivalence check.

The inference algorithm terminates when it has fully constructed the AST of P. More concretely, the number of recursive calls to InferProg is linear in the size of the given program. Critically, this number of executions is bounded by the size of the source code of P, not by the number of iterations that any loop executes. It works because any loop's iterations are independent from each other (Figure 4).

We prove Theorems 2 through 5 only for programs $P \in \mathcal{K}$ (and without reasoning about Print statements). However, the proofs rely only on the black box execution of P in $\textsc{Execute}(\boxed{P}, \sigma)$. The soundness properties therefore hold for arbitrary programs written in arbitrary languages as long as the program's externally observable behavior is equivalent to that of some program $P \in \mathcal{K}$. We will discuss these implications in Section 5.

ALGORITHM 7: Iteratively simplify code until reaching a fixed point

Input: P is a program in Prog.
Output: Succinct form of P.

```
procedure Trim(P)
    while true do
        (s, P′) ← TrimOnce(P, Nil)
        if ¬s then
            return P′
        end if
        P ← P′
    end while
end procedure
```

4.1 The Trim Transformation

This section presents the transformation that obtains $\widetilde{P}$, which we introduce in Section 3.1 to define the Konure DSL, $\mathcal{K}$ (Definition 3.4). Recall from Section 3.1 that, for any program $P \in$ Prog, $\widetilde{P}$ is the program after discarding unreachable code in P. The reachability properties facilitate the soundness proof in Section 4.5 and enable a concise way to characterize complexity in Section 4.6. In Section 5, we will extend our soundness results to programs not in $\mathcal{K}$, such as programs in Prog that may contain unreachable code.

Algorithm 7 presents the Trim transformation that obtains $\widetilde{P}$, $\widetilde{P} = \textsc{Trim}(P)$, which simplifies P by iteratively discarding unreachable branches with TrimOnce (Algorithm 8).

The TrimOnce procedure takes an initial program, $P \in$ Prog, and a path constraint, W. The procedure returns a tuple of two components. The first component is a Boolean value that indicates whether the transformation alters P. The second component is the transformed program. If P is empty, then TrimOnce returns the empty program. If the top-most nonterminal symbol of P is Seq, TrimOnce first recursively simplifies the tail of the sequence and then uses the simplified tail to construct a new Seq. If the top-most nonterminal of P is If or For, TrimOnce first recursively simplifies the subprograms and then simplifies the current control construct if possible. To perform these checks, TrimOnce updates the path constraint W and calls Solve (Section 3.3) to check reachability.

We show that the Trim transformation terminates (Theorem 6) with an equivalent program (Theorem 7) with no unreachable code (Proposition 4.24).

4.1.1 Termination of Trim. To show termination, we define a measure of code size and show that the TrimOnce transformation decreases the code size.

Definition 4.8. The *branch complexity tuple* for a program $P \in$ Prog is denoted as $\mathcal{B}(P)$ and defined as a 3-tuple of nonnegative integers, $\mathcal{B}(P) = (f, i, s) \in \mathbb{Z}^3_{\geq 0}$. Here, f denotes the number of `for`-constructs in P, i denotes the number of `if`-constructs in P, and s denotes the number of sequential queries in P:

$$\begin{aligned}
&\mathcal{B}(\epsilon) && = (0, 0, 0) && \\
&\mathcal{B}(Q\ P) && = (f_1, i_1, 1 + s_1) && \text{if } \mathcal{B}(P) = (f_1, i_1, s_1) \\
&\mathcal{B}(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2) && = (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2) && \text{if } \mathcal{B}(P_1) = (f_1, i_1, s_1), \mathcal{B}(P_2) = (f_2, i_2, s_2) \\
&\mathcal{B}(\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2) && = (1 + f_1 + f_2, i_1 + i_2, s_1 + s_2) && \text{if } \mathcal{B}(P_1) = (f_1, i_1, s_1), \mathcal{B}(P_2) = (f_2, i_2, s_2),
\end{aligned}$$

where $P, P_1, P_2 \in$ Prog, $Q \in$ Query, and $f_1, f_2, i_1, i_2, s_1, s_2 \in \mathbb{Z}_{\geq 0}$.

ALGORITHM 8: Trim unreachable branches and simplify control constructs if possible

Input: P is a program in Prog.
Input: W is a path constraint.
Output: Tuple (s, P') where $P \equiv P'$ and s indicates whether $P' \neq P$.

```
procedure TrimOnce(P, W)
    if P = ϵ then return (false, ϵ)
    else if P = Q P1 then
        (s1, P1′) ← TrimOnce(P1, W)
        return (s1, Q P1′)
    else if P = if Q then P1 else P2 then
        if Solve(W @ ⟨Q, ≥ 1, false⟩) = Unsat then return (true, Q P2)
        else if Solve(W @ ⟨Q, = 0, false⟩) = Unsat then return (true, Q P1)
        end if
        (s1, P1′) ← TrimOnce(P1, W @ ⟨Q, ≥ 1, false⟩)
        (s2, P2′) ← TrimOnce(P2, W @ ⟨Q, = 0, false⟩)
        if P1′ ≐_{W,Nil,Nil} P2′ then
            return (true, Q P1′)
        else
            return (s1 ∨ s2, if Q then P1′ else P2′)
        end if
    else if P = for Q do P1 else P2 then
        if Solve(W @ ⟨Q, ≥ 2, true⟩) = Unsat then
            return (true, if Q then P1 else P2)
        end if
        (s1, P1′) ← TrimOnce(P1, W @ ⟨Q, ≥ 1, true⟩)
        if P1′ = ϵ then
            return (true, if Q then ϵ else P2)
        end if
        if Solve(W @ ⟨Q, = 0, true⟩) = Unsat then
            s2 ← (P2 ≠ ϵ)
            P2′ ← ϵ
        else
            (s2, P2′) ← TrimOnce(P2, W @ ⟨Q, = 0, true⟩)
        end if
        return (s1 ∨ s2, for Q do P1′ else P2′)
    end if
end procedure
```

Definition 4.9. We define a partial order on $\mathbb{Z}^3_{\geq 0}$ as follows: $(f_1, i_1, s_1) \leq (f_2, i_2, s_2)$ if $f_1 \leq f_2$, $f_1 + i_1 \leq f_2 + i_2$, and $f_1 + i_1 + s_1 \leq f_2 + i_2 + s_2$.

Proof of partial order. (1) Reflexivity: For any 3-tuple $(f, i, s) \in \mathbb{Z}^3_{\geq 0}$, we have $f \leq f$, $f + i \leq f + i$, and $f + i + s \leq f + i + s$. (2) Antisymmetry: For any 3-tuples $(f_1, i_1, s_1), (f_2, i_2, s_2) \in \mathbb{Z}^3_{\geq 0}$ such that $(f_1, i_1, s_1) \leq (f_2, i_2, s_2)$ and $(f_2, i_2, s_2) \leq (f_1, i_1, s_1)$, we have $f_1 = f_2$, $f_1 + i_1 = f_2 + i_2$, and $f_1 + i_1 + s_1 = f_2 + i_2 + s_2$. (3) Transitivity: For any 3-tuples $(f_1, i_1, s_1), (f_2, i_2, s_2), (f_3, i_3, s_3) \in \mathbb{Z}^3_{\geq 0}$ such that $(f_1, i_1, s_1) \leq (f_2, i_2, s_2)$ and $(f_2, i_2, s_2) \leq (f_3, i_3, s_3)$, we have $f_1 \leq f_2 \leq f_3$, $f_1 + i_1 \leq f_2 + i_2 \leq f_3 + i_3$, and $f_1 + i_1 + s_1 \leq f_2 + i_2 + s_2 \leq f_3 + i_3 + s_3$. □

Informally, this partial order compares the code complexity of two programs. The first comparison, $f_1 \leq f_2$, compares the number of loop constructs in the programs. The second comparison,

$f_1 + i_1 \leq f_2 + i_2$, compares the total number of control constructs in the programs. The third comparison, $f_1 + i_1 + s_1 \leq f_2 + i_2 + s_2$, compares the total number of queries in the programs.

Proposition 4.10. *For any strictly decreasing sequence of branch complexity tuples* $(f_1, i_1, s_1), (f_2, i_2, s_2), \ldots \in \mathbb{Z}^3_{\geq 0}$ *such that* $(f_{k+1}, i_{k+1}, s_{k+1}) < (f_k, i_k, s_k)$ *for all* $k = 1, 2, \ldots,$ *the length of the sequence is finite.*

Proof. Since $f_1 + i_1 + s_1$ is finite, there is only a finite number of 3-tuples $(f, i, s) \in \mathbb{Z}^3_{\geq 0}$ such that $(0, 0, 0) < (f, i, s) < (f_1, i_1, s_1)$. □

Lemma 4.11. *For any program* $P \in$ *Prog and path constraint* W*, if* TrimOnce$(P, W) = (\texttt{false}, P')$*, then* $P' = P$.

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

By Algorithm 8, execution enters the branch on line 2. $P' = \epsilon = P$.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

By Algorithm 8, execution enters the branch on line 3. Since TrimOnce$(P, W) = (\texttt{false}, P')$, $s_1 = \texttt{false}$ and $P' = Q\ P_1'$. By the induction hypothesis, if $s_1 = \texttt{false},$ then $P_1' = P_1$. Hence, $P' = Q\ P_1' = Q\ P_1 = P$.

Case 3: P is of the form "If". P expands to "$\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2$", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

By Algorithm 8, execution enters the branch on line 6. Since TrimOnce$(P, W) = (\texttt{false}, P')$, execution must not enter the branches on lines 7 or 8. Execution continues after line 9. By the induction hypothesis, if $s_1 = \texttt{false},$ then $P_1' = P_1$. Also, if $s_2 = \texttt{false},$ then $P_2' = P_2$. Execution must enter the branch on line 14. Since $s_1 \vee s_2 = \texttt{false}$, $s_1 = s_2 = \texttt{false}$. Hence, $P' = \texttt{if}\ Q,\ \texttt{then}\ P_1'\ \texttt{else}\ P_2{}' = \texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2 = P$.

Case 4: P is of the form "For". P expands to "$\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2$", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 3. □

Lemma 4.12. *For any program* $P \in$ *Prog and path constraint* W*, if* TrimOnce$(P, W) = (\texttt{true}, P')$*, then* $\mathcal{B}(P') < \mathcal{B}(P)$.

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

By Algorithm 8, execution enters the branch on line 2. Hence, it is not possible to have TrimOnce$(P, W) = (\texttt{true}, P')$.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

By Algorithm 8, execution enters the branch on line 3. Since TrimOnce$(P, W) = (\texttt{true}, P')$, $s_1 = \texttt{true}$ and $P' = Q\ P_1'$. By the induction hypothesis, if $s_1 = \texttt{true},$ then $\mathcal{B}(P_1') < \mathcal{B}(P_1)$. By Definition 4.8 and Definition 4.9, $\mathcal{B}(Q\ P_1') < \mathcal{B}(Q\ P_1)$. Hence, $\mathcal{B}(P') < \mathcal{B}(P)$.

Case 3: P is of the form "If". P expands to "$\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2$", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

By Algorithm 8, execution enters the branch on line 6.

Let $f_1, f_2, i_1, i_2, s_1, s_2 \in \mathbb{Z}_{\geq 0}$ such that $\mathcal{B}(P_1) = (f_1, i_1, s_1)$ and $\mathcal{B}(P_2) = (f_2, i_2, s_2)$.

Case 3.1: Execution enters the branch on line 7.
By Definition 4.8, $\mathcal{B}(Q\ P_2) = (f_2, i_2, 1 + s_2)$. Also, $\mathcal{B}(\texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2) = (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2)$. By Definition 4.9, $(f_2, i_2, 1 + s_2) < (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2)$. Since $\textsc{TrimOnce}(P, W) = (\texttt{true}, P')$, $P' = Q\ P_2$. Hence, $\mathcal{B}(P') = \mathcal{B}(Q\ P_2) < \mathcal{B}(\texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2) = \mathcal{B}(P)$.

Case 3.2: Execution enters the branch on line 8. The proof is similar to the proof of Case 3.1.

Case 3.3: Execution continues after line 9.
By the induction hypothesis, if $s_1 = \texttt{true}$, then $\mathcal{B}(P_1') < \mathcal{B}(P_1)$. Also, if $s_2 = \texttt{true}$, then $\mathcal{B}(P_2') < \mathcal{B}(P_2)$. By Lemma 4.11, if $s_1 = \texttt{false}$, then $\mathcal{B}(P_1') = \mathcal{B}(P_1)$. Also, if $s_2 = \texttt{false}$, then $\mathcal{B}(P_2') = \mathcal{B}(P_2)$. Hence, $\mathcal{B}(P_1') \leq \mathcal{B}(P_1)$ and $\mathcal{B}(P_2') \leq \mathcal{B}(P_2)$ always hold.
If execution enters the branch on 12, since $\textsc{TrimOnce}(P, W) = (\texttt{true}, P')$, we have $P' = Q\ P_1'$. Let $f_1', i_1', s_1' \in \mathbb{Z}_{\geq 0}$ such that $\mathcal{B}(P_1') = (f_1', i_1', s_1')$. By Definition 4.8, $\mathcal{B}(Q\ P_1') = (f_1', i_1', 1 + s_1')$. Also, $\mathcal{B}(\texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2) = (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2)$. Since $\mathcal{B}(P_1') \leq \mathcal{B}(P_1)$, $(f_1', i_1', s_1') \leq (f_1, i_1, s_1)$. By Definition 4.9, $(f_1', i_1', 1 + s_1') < (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2)$. Hence, $\mathcal{B}(P') = \mathcal{B}(Q\ P_1') < \mathcal{B}(\texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2) = \mathcal{B}(P)$.
If execution enters the branch on 14, since $\textsc{TrimOnce}(P, W) = (\texttt{true}, P')$, we have $s_1 \vee s_2 = \texttt{true}$ and $P' = \texttt{if}\ Q,\ \texttt{then}\ P_1'\ \texttt{else}\ P_2{}'$. Hence, at least one of $\mathcal{B}(P_1') < \mathcal{B}(P_1)$ or $\mathcal{B}(P_2') < \mathcal{B}(P_2)$ holds. By Definition 4.8 and Definition 4.9, $\mathcal{B}(\texttt{if}\ Q,\ \texttt{then}\ P_1'\ \texttt{else}\ P_2') < \mathcal{B}(\texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2)$. Hence, $\mathcal{B}(P') < \mathcal{B}(P)$.

Case 4: P is of the form "For". P expands to "$\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2$", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 3. □

Theorem 6 (Trim Terminates). *For any program $P \in Prog$, the execution of* Trim(P) *terminates.*

Proof. By Lemma 4.12, $\mathcal{B}(P') < \mathcal{B}(P)$ on line 3 as long as $s = \texttt{true}$. Hence, the value of $\mathcal{B}(P)$ strictly decreases in each iteration of the loop as long as $s = \texttt{true}$. By Proposition 4.10, after a finite number of iterations, it is no longer possible to have $\mathcal{B}(P') < \mathcal{B}(P)$. At this time, $s = \texttt{false}$ by Lemma 4.12. Execution enters the branch on line 4. The algorithm then terminates. □

Proposition 4.13. *For any program $P \in Prog$,* Trim(Trim(P)) = Trim(P).

Proof. For any program $P_0 \in$ Prog, let $P_1 = \textsc{Trim}(P_0)$. In the last iteration of the loop in Algorithm 7, variable $s = \texttt{false}$ on line 3. By Lemma 4.11, variable $P' = P$ at this time. Since the algorithm returns P_1, we have $\textsc{TrimOnce}(P_1, \text{Nil}) = (\texttt{false}, P_1)$. Let $P_2 = \textsc{Trim}(P_1)$. In the first iteration of the loop in Algorithm 7, variable $s = \texttt{false}$ and $P' = P = P_1$ on line 3. Hence, the return value $P_2 = P_1$. In other words, $\textsc{Trim}(\textsc{Trim}(P_0)) = \textsc{Trim}(P_0)$. □

4.1.2 *Soundness of* Trim.

To show that Trim produces an equivalent program, we show that each recursive call to TrimOnce rewrites each subprogram into a corresponding subprogram that is equivalent with respect to a path constraint.

Definition 4.14. We define the *observational equivalence* relation on Prog as follows: $P_1 \equiv P_2$ if for any context $\sigma \in Context$, $\sigma(P_1) = \sigma(P_2)$.

Proof of equivalence relation. (1) Reflexivity: For any program $P \in$ Prog, $\sigma(P) = \sigma(P)$ for all $\sigma \in Context$. (2) Symmetry: For any programs $P_1, P_2 \in$ Prog such that $P_1 \equiv P_2$, $\sigma(P_2) = \sigma(P_1)$ for all $\sigma \in Context$. (3) Transitivity: For any programs $P_1, P_2, P_3 \in$ Prog such that $P_1 \equiv P_2$ and $P_2 \equiv P_3$, $\sigma(P_1) = \sigma(P_2) = \sigma(P_3)$ for all $\sigma \in Context$. □

Proposition 4.15. *For any programs $P_1, P_2 \in Prog$, if $P_1 \equiv P_2$, then* $\textsc{Infer}(\boxed{P_1}) = \textsc{Infer}(\boxed{P_2})$.

Proof. By Definition 4.14, $\sigma(P_1) = \sigma(P_2)$ for any context $\sigma \in Context$. By Definition 3.9, for any σ, we have $\textsc{Execute}(\boxed{P_1}, \sigma) = \textsc{Execute}(\boxed{P_2}, \sigma)$. By Algorithm 1, $\textsc{Infer}(\boxed{P_1}) = \textsc{Infer}(\boxed{P_2})$. □

Definition 4.16. For any path constraint W, we define a relation on Prog as follows: $P_1 \equiv_{W,Y_1,Y_2} P_2$ if for any context $\sigma \in Context$ that satisfies W, $\sigma(P_1') = \sigma(P_2)$, where P_1' is the program obtained from P_1 after replacing all occurrences of variables in Y_1 with their counterparts in Y_2.

Proposition 4.17. *For any programs $P_1, P_2 \in Prog$ and list of variables $Y \in \overline{Variable}$, $P_1 \equiv_{Nil,Nil,Nil} P_2$ if and only if $P_1 \equiv_{Nil,Y,Y} P_2$.*

Proof. By definition. □

Proposition 4.18. *For any programs $P_1, P_2 \in Prog$, $P_1 \equiv_{Nil,Nil,Nil} P_2$ if and only if $P_1 \equiv P_2$.*

Proof. By definition. □

Proposition 4.19. *For any programs $P_1, P_2 \in Prog$, path constraint W, and lists of variables $Y_1, Y_2 \in \overline{Variable}$, if $P_1 \doteq_{W,Y_1,Y_2} P_2$ then $P_1 \equiv_{W,Y_1,Y_2} P_2$.*

Proof. By induction on the derivation of P_1, P_2, using Definition 3.18 and Figure 15. □

Proposition 4.20. *For any programs $P_1, P_2 \in Prog$, if $P_1 \doteq P_2$, then $P_1 \equiv P_2$.*

Proof. By Proposition 4.18 and Proposition 4.19. □

Lemma 4.21. *For any program $P \in Prog$ and path constraint W, if $\textsc{TrimOnce}(P, W) = (s, P')$, then $P' \equiv_{W,Nil,Nil} P$.*

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

By Algorithm 8, execution enters the branch on line 2. Hence, $P' = \epsilon = P$. By Definition 4.16, $P' \equiv_{W,\text{Nil},\text{Nil}} P$.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

By Algorithm 8, execution enters the branch on line 3. By the induction hypothesis, variable P_1' satisfies $P_1' \equiv_{W,\text{Nil},\text{Nil}} P_1$. For any context $\sigma \in Context$ that satisfies W, let $\sigma_1 = \sigma[Q.y \mapsto \sigma(Q)]$. Since σ_1 only adds the mapping of a new variable $Q.y$ that does not appear in W, σ_1 also satisfies W. By Definition 4.16, $\sigma_1(P_1') = \sigma_1(P_1)$. By Figure 12, $\sigma(Q\ P_1') = \sigma(Q\ P_1)$. By Definition 4.16, $Q\ P_1' \equiv_{W,\text{Nil},\text{Nil}} Q\ P_1$. Since $\textsc{TrimOnce}(P, W) = (s, P')$, $P' = Q\ P_1'$. Hence, $P' \equiv_{W,\text{Nil},\text{Nil}} P$.

Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

By Algorithm 8, execution enters the branch on line 6.

Case 3.1: $\textsc{Solve}(W \mathbin{@} \langle Q, \geq 1, \texttt{false}\rangle)$ is unsatisfiable.

Execution enters the branch on line 7. By Proposition 4.6, for any context $\sigma \in Context$ that satisfies W, $\sigma(Q) = \emptyset$, because $|\sigma(Q)| \geq 1$ is impossible. Hence, $\sigma[Q.y \mapsto \sigma(Q)] = \sigma$. By Figure 12, $\sigma(Q\ P_2) = \sigma(\texttt{if}\ Q,\ \texttt{then}\ P_1\ \texttt{else}\ P_2)$. By Definition 4.16, $Q\ P_2 \equiv_{W,\text{Nil},\text{Nil}}$ `if` Q, `then` P_1 `else` P_2. Since $\textsc{TrimOnce}(P, W) = (s, P')$, $P' = Q\ P_2$. Hence, $P' \equiv_{W,\text{Nil},\text{Nil}} P$.

Case 3.2: $\textsc{Solve}(W @ \langle Q, = 0, \texttt{false}\rangle)$ is unsatisfiable.
Execution enters the branch on line 8. The proof is similar to the proof of Case 3.1.

Case 3.3: Both $\textsc{Solve}(W @ \langle Q, \geq 1, \texttt{false}\rangle)$ and $\textsc{Solve}(W @ \langle Q, = 0, \texttt{false}\rangle)$ are satisfiable.
Execution continues after line 9. By the induction hypothesis, variables P_1' and P_2' satisfy $P_1' \equiv_{W @ \langle Q, \geq 1, \texttt{false}\rangle, \text{Nil}, \text{Nil}} P_1$ and $P_2' \equiv_{W @ \langle Q, =0, \texttt{false}\rangle, \text{Nil}, \text{Nil}} P_2$.
For any context $\sigma \in \mathit{Context}$ that satisfies W, only one of $|\sigma(Q)| = 0$ and $|\sigma(Q)| \geq 1$ holds.

Case 3.3.1: If $|\sigma(Q)| = 0$, then σ satisfies $W @ \langle Q, = 0, \texttt{false}\rangle$.
By Definition 4.16, $\sigma(P_2') = \sigma(P_2)$. Since $\sigma(Q) = \emptyset$, $\sigma[Q.y \mapsto \sigma(Q)] = \sigma$. By Figure 12, $\sigma(\texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_2') = \sigma(\texttt{if } Q\texttt{, then } P_1 \texttt{ else } P_2)$.

Case 3.3.2: If $|\sigma(Q)| \geq 1$, then σ satisfies $W @ \langle Q, \geq 1, \texttt{false}\rangle$.
Let $\sigma_1 = \sigma[Q.y \mapsto \sigma(Q)]$. Since σ_1 only adds the mapping of a new variable $Q.y$ that does not appear in W, σ_1 also satisfies the path constraint $W @ \langle Q, \geq 1, \texttt{false}\rangle$. By Definition 4.16, $\sigma_1(P_1') = \sigma_1(P_1)$. By Figure 12, $\sigma(\texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_2') = \sigma(\texttt{if } Q\texttt{, then } P_1 \texttt{ else } P_2)$.

Either case, we have $\sigma(\texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_2') = \sigma(\texttt{if } Q\texttt{, then } P_1 \texttt{ else } P_2)$. By Definition 4.16, $\texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_2' \equiv_{W, \text{Nil}, \text{Nil}} \texttt{if } Q\texttt{, then } P_1 \texttt{ else } P_2$.
Next, consider if P_1' and P_2' are identical except for equivalent variables.

Case 3.3.1: If $P_1' \doteq_{W, \text{Nil}, \text{Nil}} P_2'$, then execution enters the branch on line 12. By Proposition 4.19, $P_1' \equiv_{W, \text{Nil}, \text{Nil}} P_2'$. By Definition 4.16 and Figure 12, we have $\texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_1' \equiv_{W, \text{Nil}, \text{Nil}} \texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_2'$.
Since $\textsc{TrimOnce}(P, W) = (s, P')$, $P' = Q\ \ P_1'$. Clearly, $Q\ \ P_1' \equiv_{W, \text{Nil}, \text{Nil}} \texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_1'$. By Definition 4.16, $P' \equiv_{W, \text{Nil}, \text{Nil}} P$.

Case 3.3.2: If $P_1' \not\doteq_{W, \text{Nil}, \text{Nil}} P_2'$, execution enters the branch on line 14. Since $\textsc{TrimOnce}(P, W) = (s, P')$, we have $P' = \texttt{if } Q\texttt{, then } P_1' \texttt{ else } P_2'$. Hence, $P' \equiv_{W, \text{Nil}, \text{Nil}} P$.

Case 4: P is of the form "For". P expands to "$\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2$", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 3. □

Lemma 4.22. *For any program $P \in \mathit{Prog}$, if $\textsc{TrimOnce}(P, \mathit{Nil}) = (s, P')$, then $P' \equiv P$.*

Proof. By Lemma 4.21, $P' \equiv_{\text{Nil}, \text{Nil}, \text{Nil}} P$. By Proposition 4.18, $P' \equiv P$. □

Theorem 7 (Trim Preserves Semantics). *For any program $P \in \mathit{Prog}$, $\textsc{Trim}(P) \equiv P$.*

Proof. By Lemma 4.22, $P' \equiv P$ on line 3 in each iteration of the loop. By Theorem 6, the loop terminates. By Definition 4.14, the final program P' preserves the semantics of the initial program. □

We next outline the reachability properties of the simplified program. Intuitively, since TrimOnce discards unreachable branches, the remaining branches are all reachable.

Proposition 4.23. *For any program $P \in \mathit{Prog}$ and path constraint W, if $\textsc{TrimOnce}(P, W) = (\texttt{false}, P')$, then the following hold for P':*

- *For any query $Q \in \mathit{Query}$ in P', there exists a context $\sigma \in \mathit{Context}$ such that Q is used while evaluating $\sigma(P')$.*
- *For any `if`-construct "`if` $Q \ldots$" in P', there exists a context $\sigma \in \mathit{Context}$ such that Q is used while evaluating $\sigma(P')$ and the corresponding row count $r \geq 1$.*

- *For any* `if`*-construct "*`if` *$Q \ldots$" in P', there exists a context $\sigma \in$ Context such that Q is used while evaluating $\sigma(P')$ and the corresponding row count $r = 0$.*
- *For any* `for`*-construct "*`for` *$Q \ldots$" in P', there exists a context $\sigma \in$ Context such that Q is used while evaluating $\sigma(P')$ and the corresponding row count $r \geq 2$.*

RATIONALE. By induction on the derivation of P', using Proposition 4.6. □

PROPOSITION 4.24 (REACHABILITY). *For any program $P \in$ Prog, the following hold:*

- *For any query $Q \in$ Query in TRIM(P), there exists a context $\sigma \in$ Context such that Q is used while evaluating $\sigma($TRIM$(P))$.*
- *For any* `if`*-construct "*`if` *$Q \ldots$" in TRIM(P), there exists a context $\sigma \in$ Context such that Q is used while evaluating $\sigma($TRIM$(P))$ and the corresponding row count $r \geq 1$.*
- *For any* `if`*-construct "*`if` *$Q \ldots$" in TRIM(P), there exists a context $\sigma \in$ Context such that Q is used while evaluating $\sigma($TRIM$(P))$ and the corresponding row count $r = 0$.*
- *For any* `for`*-construct "*`for` *$Q \ldots$" in TRIM(P), there exists a context $\sigma \in$ Context such that Q is used while evaluating $\sigma($TRIM$(P))$ and the corresponding row count $r \geq 2$.*

RATIONALE. In the last iteration of the loop in Algorithm 7, variable $s =$ `false` on line 3. The rest of the proof follows from Proposition 4.23. □

4.2 Source Code Characteristics

This section defines the functions $\mathcal{T}(\cdot)$, $\mathcal{R}(\cdot)$, and $\mathcal{D}(\cdot)$, which we introduce in Section 3.1 to define the KONURE DSL. To present the KONURE DSL restrictions formally, we define the following characteristics for describing the source code of a program in Prog:

Definition 4.25. For any program $P \in$ Prog, function $\mathcal{F}(P)$ returns the first query of P if P is nonempty or Nil if P is empty:

$$\begin{aligned}
\mathcal{F}(\epsilon) &= \text{Nil} \\
\mathcal{F}(Q\ P) &= Q \\
\mathcal{F}(\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2) &= Q \\
\mathcal{F}(\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2) &= Q,
\end{aligned}$$

where $P, P_1, P_2 \in$ Prog and $Q \in$ Query.

Definition 4.26. Let SQuery be the set of skeleton queries (Appendix A). For any program $P \in$ Prog and $q \in$ SQuery $\cup$ { Nil }, function $\mathcal{C}(q, P)$ returns the number of queries in P that share the skeleton q:

$$\begin{aligned}
\mathcal{C}(\text{Nil}, P) &= 0 \\
\mathcal{C}(q, \epsilon) &= 0 \\
\mathcal{C}(q, Q\ P) &= \begin{cases} 1 + \mathcal{C}(q, P) & \text{if } \pi_{\mathbb{S}} Q = q \\ \mathcal{C}(q, P) & \text{otherwise} \end{cases}, \quad (q \neq \text{Nil}) \\
\mathcal{C}(q, \texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2) &= \begin{cases} 1 + \mathcal{C}(q, P_1) + \mathcal{C}(q, P_2) & \text{if } \pi_{\mathbb{S}} Q = q \\ \mathcal{C}(q, P_1) + \mathcal{C}(q, P_2) & \text{otherwise} \end{cases}, \quad (q \neq \text{Nil}) \\
\mathcal{C}(q, \texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2) &= \begin{cases} 1 + \mathcal{C}(q, P_1) + \mathcal{C}(q, P_2) & \text{if } \pi_{\mathbb{S}} Q = q \\ \mathcal{C}(q, P_1) + \mathcal{C}(q, P_2) & \text{otherwise} \end{cases}, \quad (q \neq \text{Nil}),
\end{aligned}$$

where $P, P_1, P_2 \in$ Prog, $Q \in$ Query, and $q \in$ SQuery $\cup$ { Nil }.

Definition 4.27. For any program $P \in$ Prog, function $\mathcal{R}(P)$ returns the set of all queries in P whose immediate subsequent query on the nonempty branch shares skeleton with other subsequent queries:

$$\mathcal{R}(\epsilon) = \emptyset$$
$$\mathcal{R}(Q\ P) = \begin{cases} \{Q.y\} \cup \mathcal{R}(P) & \text{if } \mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P), P) \geq 2 \\ \mathcal{R}(P) & \text{otherwise} \end{cases}$$
$$\mathcal{R}(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2) = \begin{cases} \{Q.y\} \cup \mathcal{R}(P_1) \cup \mathcal{R}(P_2) & \text{if } \mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) \geq 2 \\ \mathcal{R}(P_1) \cup \mathcal{R}(P_2) & \text{otherwise} \end{cases}$$
$$\mathcal{R}(\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2) = \begin{cases} \{Q.y\} \cup \mathcal{R}(P_1) \cup \mathcal{R}(P_2) & \text{if } \mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) \geq 2 \\ \mathcal{R}(P_1) \cup \mathcal{R}(P_2) & \text{otherwise,} \end{cases}$$

where $P, P_1, P_2 \in$ Prog and $Q \in$ Query.

Definition 4.28. For any program $P \in$ Prog and context $\sigma \in$ *Context*, $\mathcal{T}_\sigma(P)$ denotes the set of queries that each returns at least two rows when executing P using σ:

$$\mathcal{T}_\sigma(\epsilon) = \emptyset$$
$$\mathcal{T}_\sigma(Q\ P) = \begin{cases} \{Q.y\} \cup \mathcal{T}_{\sigma[Q.y \mapsto \sigma(Q)]}(P) & \text{if } |\sigma(Q)| \geq 2 \\ \mathcal{T}_{\sigma[Q.y \mapsto \sigma(Q)]}(P) & \text{otherwise} \end{cases}$$
$$\mathcal{T}_\sigma(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2) = \begin{cases} \{Q.y\} \cup \mathcal{T}_{\sigma[Q.y \mapsto \sigma(Q)]}(P_1) & \text{if } |\sigma(Q)| \geq 2 \\ \mathcal{T}_{\sigma[Q.y \mapsto \sigma(Q)]}(P_1) & \text{if } |\sigma(Q)| = 1 \\ \mathcal{T}_\sigma(P_2) & \text{otherwise} \end{cases}$$
$$\mathcal{T}_\sigma(\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2) = \begin{cases} \{Q.y\} \cup \bigcup_{i=1}^{r} \mathcal{T}_{\sigma[Q.y \mapsto \sigma(Q)[i]]}(P_1) & \text{if } |\sigma(Q)| = r \geq 2 \\ \mathcal{T}_{\sigma[Q.y \mapsto \sigma(Q)]}(P_1) & \text{if } |\sigma(Q)| = 1 \\ \mathcal{T}_\sigma(P_2) & \text{otherwise,} \end{cases}$$

where $P, P_1, P_2 \in$ Prog, $Q \in$ Query, $y \in$ *Variable*, and $r \in \mathbb{Z}_{\geq 0}$.

Definition 4.29. For any program $P \in$ Prog, function $\mathcal{T}(P)$ returns the set of queries in P that may retrieve at least two rows during any execution: $\mathcal{T}(P) = \bigcup_{\sigma \in Context} \mathcal{T}_\sigma(P)$.

Definition 4.30. For any program $P \in$ Prog, predicate $\mathcal{D}(P)$ is `true` if and only if the two branches of any conditional statement in P start with queries with different skeletons:

$$\mathcal{D}(\epsilon) = \texttt{true}$$
$$\mathcal{D}(Q\ P) = \mathcal{D}(P)$$
$$\mathcal{D}(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2) = \begin{cases} \texttt{true} & \text{if } P_1 = P_2 = \epsilon \\ \pi_{\mathcal{S}}\mathcal{F}(P_1) \neq \pi_{\mathcal{S}}\mathcal{F}(P_2) \wedge \mathcal{D}(P_1) \wedge \mathcal{D}(P_2) & \text{otherwise} \end{cases}$$
$$\mathcal{D}(\texttt{for } Q \texttt{ do } P_1 \texttt{ else } P_2) = \begin{cases} \texttt{true} & \text{if } P_1 = P_2 = \epsilon \\ \mathcal{D}(P_1) \wedge \mathcal{D}(P_2) & \text{otherwise,} \end{cases}$$

where $P, P_1, P_2 \in$ Prog and $Q \in$ Query.

4.3 Soundness of DetectLoops

We show that the outcome of DetectLoops (Algorithm 3) is consistent with the loop layout tree obtained from the source code (Theorem 1). To facilitate discussion, we define an auxiliary procedure, DetectLoopsAux (Algorithm 9). This procedure is the same as DetectLoops except for

ALGORITHM 9: Loop detection algorithm (Algorithm 3) with auxiliary variables

Input: e is either Nil or a nonempty list of query-result pairs $(Q_1, r_1), \ldots, (Q_n, r_n)$.
Input: $P \in \text{Prog}$ is an auxiliary variable used only in the soundness proof.
Input: $\sigma \in$ *Context* is an auxiliary variable used only in the soundness proof.
Output: Loop layout tree constructed from e.

1: **procedure** DetectLoopsAux(e, P, σ)
2:     **if** e = Nil **then**
3:         **return** Nil
4:     **end if**
5:     $(Q_1, r_1), \ldots, (Q_n, r_n) \leftarrow e$
6:     $a \leftarrow$ empty list
7:     **for** $j \leftarrow 2, 3, \ldots, n$ **do** ▷ Identify repetitions
8:         **if** $\pi_{\mathbb{S}} Q_j = \pi_{\mathbb{S}} Q_2$ **then**
9:             Append j to a
10:         **end if**
11:     **end for**
12:     **if** $r_1 \leq 1$ **or** $r_1 \neq \texttt{len}(a)$ **then** ▷ Did not find repetitions caused by any loops that iterate over Q_1
13:         $e' \leftarrow (Q_2, r_2), \ldots, (Q_n, r_n)$
14:         $\sigma' \leftarrow \sigma[Q_1.y \mapsto \sigma(Q_1)]$
15:         **if** $r_1 = 0$ **then**
16:             $P' \leftarrow$ Subprogram of P in the empty branch
17:         **else**
18:             $P' \leftarrow$ Subprogram of P in the nonempty branch
19:         **end if**
20:         $l \leftarrow$ DetectLoopsAux(e', P', σ')
21:         **return** $(Q_1, r_1) \searrow l$
22:     **else** ▷ Found a loop that iterates over Q_1
23:         Append $n + 1$ to a
24:         **for** $j \leftarrow 1, 2, \ldots, r_1$ **do**
25:             $b \leftarrow a[j]$
26:             $c \leftarrow a[j + 1] - 1$
27:             $e' \leftarrow (Q_b, r_b), \ldots, (Q_c, r_c)$
28:             $\sigma' \leftarrow \sigma[Q_1.y \mapsto \sigma(Q_1)[j]]$
29:             $P' \leftarrow$ Subprogram of P in the nonempty branch
30:             $l_j \leftarrow$ DetectLoopsAux(e', P', σ')
31:         **end for**
32:         **return** $(Q_1, r_1) \circlearrowleft (l_1, \ldots, l_{r_1})$
33:     **end if**
34: **end procedure**

two additional variables, P and σ, which are used in the proof but do not affect the results of the algorithm.

Lemma 4.31. *For any program $P_0 \in \mathcal{K}$ and context $\sigma_0 \in$ Context if $\sigma_0 \vdash P_0 \Downarrow_{exec} e_0$, during the calculation of* DetectLoopsAux(e_0, P_0, σ_0), *if the parameters of a recursive call* DetectLoopsAux(e, P, σ) *satisfy $\sigma \vdash P \Downarrow_{exec} e$ and Algorithm 9 enters line 5 then:*

(1) $\mathcal{F}(P) = Q_1$,
(2) $|\sigma(Q_1)| = r_1$, and
(3) if $r_1 \geq 2$ then $Q_1.y \in \mathcal{T}(P_0)$.

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$. Since $\sigma \vdash P \Downarrow_{\text{exec}} e$, by Figure 16, $e = \text{Nil}$. Algorithm 9 returns before line 5.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol. Since $\sigma \vdash P \Downarrow_{\text{exec}} e$, by Figure 16, $Q_1 = Q$ and $r_1 = |\sigma(Q)| = |\sigma(Q_1)|$. By Definition 4.25, $\mathcal{F}(P) = Q = Q_1$. By Definition 4.28, if $r_1 \geq 2$, then $Q_1.y \in \mathcal{T}_\sigma(P)$ and $Q_1.y \in \mathcal{T}_{\sigma_0}(P_0)$. By Definition 4.29, $Q_y.y \in \mathcal{T}(P_0)$.

Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 2.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 2. □

Lemma 4.32. *For any program $P_0 \in \mathcal{K}$ and context $\sigma_0 \in Context$ if $\sigma_0 \vdash P_0 \Downarrow_{exec} e_0$, during the calculation of* DetectLoopsAux(e_0, P_0, σ_0), *if the parameters of a recursive call* DetectLoopsAux(e, P, σ) *satisfy $\sigma \vdash P \Downarrow_{exec} e$ and $\sigma \vdash P \Downarrow_{loops} l$, then* DetectLoopsAux$(e, P, \sigma) = l$.

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$. Since $\sigma \vdash P \Downarrow_{\text{exec}} e$, by Figure 16, $e = \text{Nil}$. Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, $l = \text{Nil}$. By Algorithm 9, DetectLoopsAux$(\text{Nil}, P, \sigma) = \text{Nil}$.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

Let $r = |\sigma(Q)|$ and $\sigma_1 = \sigma[Q.y \mapsto \sigma(Q)]$.

Since $\sigma \vdash P \Downarrow_{\text{exec}} e$, by Figure 16, there exists a list of query-result pairs e_1 such that $e = (Q, r) @ e_1$ and $\sigma_1 \vdash P_1 \Downarrow_{\text{exec}} e_1$. By Lemma 4.31, $e = (Q_1, r_1) @ e_1$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, there exists a loop layout tree l_1 such that $l = (Q, r) \searrow l_1$ and $\sigma_1 \vdash P_1 \Downarrow_{\text{loops}} l_1$. By Lemma 4.31, $l = (Q_1, r_1) \searrow l_1$.

Case 2.1: $r \leq 1$. In Algorithm 9, execution enters the branch on line 12.

Case 2.2: $r \geq 2$. By Lemma 4.31, $Q.y \in \mathcal{T}(P_0)$. By Definition 3.4, $Q.y \notin \mathcal{R}(P_0)$. By Definition 4.27, $\mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) \leq 1$. By Figure 16, $\pi_{\mathcal{S}}\mathcal{F}(P_1)$ appears at most once in e_1. In Algorithm 9, the branch on line 8 never executes, so the length of a in the procedure remains zero. Execution enters the branch on line 12.

In both cases, by Lemma 4.31, variable $\sigma' = sigma_1$. Also, variables $e' = e_1$ and $P' = P_1$. Algorithm 9 recursively calls DetectLoopsAux(e_1, P_1, σ_1) on line 20, which returns l_1 by the induction hypothesis. Hence, DetectLoopsAux$(e, P, \sigma) = (Q_1, r_1) \searrow$DetectLoopsAux$(e_1, P_1, \sigma_1) = (Q, r) \searrow l_1 = l$.

Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Case 3.1: $\sigma(Q) = \emptyset$.

Since $\sigma \vdash P \Downarrow_{\text{exec}} e$, by Figure 16, there exists a list of query-result pairs e_2 such that $e = (Q, 0) @ e_2$ and $\sigma \vdash P_2 \Downarrow_{\text{exec}} e_2$. By Lemma 4.31, $e = (Q_1, r_1) @ e_2$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, there exists a loop layout tree l_2 such that $l = (Q, 0) \searrow l_2$ and $\sigma \vdash P_2 \Downarrow_{\text{loops}} l_2$. By Lemma 4.31, $l = (Q_1, r_1) \searrow l_2$.

By Lemma 4.31, $r_1 = |\sigma(Q)| = 0$. In Algorithm 9, execution enters the branch on line 12. By Definition 3.7, variable $\sigma' = \sigma$. Also, variables $e' = e_2$ and $P' = P_2$. Algorithm 9 recursively calls DetectLoopsAux(e_2, P_2, σ) on line 20, which returns l_2 by the induction hypothesis. Hence, DetectLoopsAux$(e, P, \sigma) = (Q_1, r_1) \searrow$ DetectLoopsAux$(e_2, P_2, \sigma) = (Q, 0) \searrow l_2 = l$.

Case 3.2: $\sigma(Q) \neq \emptyset$. The proof is similar to the proof of Case 2.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Case 4.1: $\sigma(Q) = \emptyset$. The proof is similar to the proof of Case 3.1.

Case 4.2: $|\sigma(Q)| = 1$. The proof is similar to the proof of Case 2.

Case 4.3: $|\sigma(Q)| = r \geq 2$.

Let $x_1, \ldots, x_r$ be the rows of $\sigma(Q)$, $\sigma(Q) = (x_1, \ldots, x_r)$. For $i = 1, \ldots, r$, let $\sigma_i = \sigma[Q.y \mapsto x_i]$.

Since $\sigma \vdash P \Downarrow_{\text{exec}} e$, by Figure 16, there exists lists of query-result pairs $e_1, \ldots, e_r$ such that $e = (Q, r) @ e_1 @ \ldots @ e_r$ and $\sigma_i \vdash P_1 \Downarrow_{\text{exec}} e_i$ for each $i = 1, \ldots, r$. By Lemma 4.31, $r = r_1$ and $e = (Q_1, r_1) @ e_1 @ \ldots @ e_{r_1}$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, there exists loop layout trees $l_1, \ldots, l_r$ such that $l = (Q, r) \circlearrowright (l_1, \ldots, l_r)$ and $\sigma_i \vdash P_1 \Downarrow_{\text{loops}} l_i$ for each $i = 1, \ldots, r$.

Since $r \geq 2$, by Lemma 4.31, $Q.y \in \mathcal{T}(P_0)$. By Definition 3.4, $Q.y \notin \mathcal{R}(P_0)$. By Definition 4.27, $\mathcal{C}(\pi_{\mathcal{S}} \mathcal{F}(P_1), P_1) \leq 1$.

By Definition 3.4 and Algorithm 8, $P_1 \neq \epsilon$. By Definition 4.26, $\mathcal{C}(\pi_{\mathcal{S}} \mathcal{F}(P_1), P_1) \geq 1$. Hence, $\mathcal{C}(\pi_{\mathcal{S}} \mathcal{F}(P_1), P_1) = 1$.

By Figure 16, $\pi_{\mathcal{S}} \mathcal{F}(P_1)$ appears in the first query-result pair in each e_i $(i = 1, \ldots, r_1)$ and not in any other query-result pairs. In Algorithm 9, the branch on line 8 executes if and only if the query under inspection comes from the first query-result pair of any e_i $(i = 1, \ldots, r_1)$. Hence, the length of a equals r_1 on line 12. Execution does not enter the branch on this line.

Execution continues to the loop on line 24. In the ith iteration of this loop, variable b is the index of the first query-result pair of e_i and variable c is the index of the last query-result pair of e_i $(i = 1, \ldots, r_1)$. By Lemma 4.31, variable $\sigma' = \sigma[Q_1.y \mapsto x_i] = \sigma_i$. Also, variables $e' = e_i$ and $P' = P_1$. Algorithm 9 recursively calls DetectLoopsAux(e_i, P_1, σ_i) on line 30, which returns l_i by the induction hypothesis. Hence, DetectLoopsAux$(e, P, \sigma) = (Q_1, r_1) \circlearrowright ($DetectLoopsAux$(e_1, P_1, \sigma_1), \ldots,$ DetectLoopsAux$(e_r, P_1, \sigma_{r_1})) = (Q, r) \circlearrowright (l_1, \ldots, l_r) = l$. □

Theorem 1 (Loop Detection). *For any program $P \in \mathcal{K}$ and context $\sigma \in$ Context, if $\sigma \vdash P \Downarrow_{exec} e$ and $\sigma \vdash P \Downarrow_{loops} l$, then* DetectLoops$(e) = l$.

Proof. By Lemma 4.32, DetectLoopsAux$(e, P, \sigma) = l$. Since Algorithm 9 and Algorithm 3 differ only in the auxiliary variables, DetectLoops$(e) = l$. □

4.4 Soundness of GetTrace

We show that the outcome of MatchPath corresponds to a path through the program's abstract syntax tree (Lemma 4.41) and corresponds to a path through the loop layout tree (Lemma 4.41). We then show the soundness of GetTrace (Theorem 2).

To facilitate discussion, we first introduce notation for reasoning about subtrees of the program AST (Section 4.4.1) and subtrees of the loop layout tree (Section 4.4.2).

4.4.1 Traversing the Program AST.

Proposition 4.33. *For any programs $P_1, P_2, P_3 \in$ Prog and annotated traces t_1, t_2:*

(1) *if* $P_1 \xrightarrow{t_1} P_2$ *and* $P_2 \xrightarrow{t_2} P_3$, *then* $P_1 \xrightarrow{t_1 \,@\, t_2} P_3$.
(2) *if* $P_1 \xrightarrow{t_1} P_2$ *and* $P_1 \xrightarrow{t_1 \,@\, t_2} P_3$, *then* $P_2 \xrightarrow{t_2} P_3$.

Proof. By induction on the length of t_1 and the derivation of P_1. □

Remark. Note that the reverse direction of subtraction does not hold. If $P_2 \xrightarrow{t_2} P_3$ and $P_1 \xrightarrow{t_1 \,@\, t_2} P_3$, then $P_1 \xrightarrow{t_1} P_2$ may not hold. Consider the following programs:

$$
\begin{aligned}
P_1 &= Q_1\ Q_2,\\
P_2 &= \texttt{if}\ Q_2\ \texttt{then}\ Q_3\ \texttt{else}\ \epsilon,\\
P_3 &= \epsilon.
\end{aligned}
$$

Let $t_1 = \langle Q_1, 0, \text{NotLoop}\rangle$ and $t_2 = \langle Q_2, 0, \text{NotLoop}\rangle$. By Figure 18, $P_2 \xrightarrow{t_2} \epsilon$, $P_1 \xrightarrow{t_1 \,@\, t_2} \epsilon$, and $P_1 \xrightarrow{t_1} Q_2$. However, $Q_2 \neq P_2$.

Proposition 4.34. *For any programs $P_1, P_2 \in Prog$ and annotated traces t_1, t_2, if $P_1 \xrightarrow{t_1 \,@\, t_2} P_2$, then there exists program $P_3 \in Prog$ such that $P_1 \xrightarrow{t_1} P_3$.*

Proof. The proof is by induction on the length of t_1 and the derivation of P_1.

<u>Case 1:</u> $t_1 = \text{Nil}$. Let $P_3 = P_1$. By Figure 18, $P_1 \xrightarrow{\text{Nil}} P_1$.
<u>Case 2:</u> $t_1 = \langle Q', r, \lambda\rangle \,@\, t_1'$. We have $t_1 \,@\, t_2 = \langle Q', r, \lambda\rangle \,@\, t_1' \,@\, t_2$.

<u>Case 2.1:</u> $P_1 = \epsilon$. Since $t_1 \,@\, t_2 \neq \text{Nil}$, it is not possible to have $P_1 \xrightarrow{t_1 \,@\, t_2} P_2$ by Figure 18.
<u>Case 2.2:</u> P_1 is of the form “Seq”. P_1 expands to “$Q\ P_1'$”, where Q corresponds to the Query symbol and P_1' corresponds to the Prog symbol.

Since $P_1 \xrightarrow{t_1 \,@\, t_2} P_2$, by Figure 18, $Q \doteq Q'$ and $P_1' \xrightarrow{t_1' \,@\, t_2} P_2$. By the induction hypothesis, there exists program $P_3 \in \text{Prog}$ such that $P_1' \xrightarrow{t_1'} P_3$. By Figure 18, $P_1 \xrightarrow{\langle Q', r, \lambda\rangle} P_1'$. By Proposition 4.33, $P_1 \xrightarrow{t_1} P_3$.

<u>Case 2.3:</u> P is of the form “If”. P expands to “$\texttt{if}\ Q,\ \texttt{then}\ P_1'\ \texttt{else}\ P_2'$”, where Q corresponds to the Query symbol, P_1' corresponds to the first Prog symbol, and P_2' corresponds to the second Prog symbol.

<u>Case 2.3.1:</u> $r > 0$. The proof is similar to the proof of Case 2.2.
<u>Case 2.3.2:</u> $r = 0$.

Since $P_1 \xrightarrow{t_1 \,@\, t_2} P_2$, by Figure 18, $Q \doteq Q'$ and $P_2' \xrightarrow{t_1' \,@\, t_2} P_2$.
By the induction hypothesis, there exists program $P_3 \in \text{Prog}$ such that $P_2' \xrightarrow{t_1'} P_3$. By Figure 18, $P_1 \xrightarrow{\langle Q', r, \lambda\rangle} P_2'$. By Proposition 4.33, $P_1 \xrightarrow{t_1} P_3$.

<u>Case 2.4:</u> P is of the form “For”. P expands to “$\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2$”, where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 2.3. □

Proposition 4.35. *For any program $P \in Prog$, context $\sigma \in Context$, loop layout tree l such that $\sigma \vdash P \Downarrow_{loops} l$, and annotated trace $t \in$ GetAnnotatedTrace(l), we have $P \xrightarrow{t} \epsilon$.*

Proof. This proof is by induction on the derivation of P.

<u>Case 1:</u> $P = \epsilon$.

By Figure 17, $\sigma \vdash P \Downarrow_{\text{loops}}$ Nil. By Algorithm 4, GetAnnotatedTrace(Nil) = { Nil }. Hence, t = Nil. By Figure 18, $\epsilon \xrightarrow{\text{Nil}} \epsilon$.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

Let $\sigma_1 = \sigma[Q.y \mapsto \sigma(Q)]$ and $r = |\sigma(Q)|$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, there exists a loop layout tree l_1 such that $l = (Q, r) \searrow l_1$ and $\sigma_1 \vdash P_1 \Downarrow_{\text{loops}} l_1$. By Algorithm 4, GetAnnotatedTrace(l) = $\{ \langle Q, r, \text{NotLoop} \rangle \mathbin{@} t' \mid t' \in$ GetAnnotatedTrace$(l_1) \}$.

Since $t \in$ GetAnnotatedTrace(l), there exists $t' \in$ GetAnnotatedTrace(l_1) such that $t = \langle Q, r, \text{NotLoop} \rangle \mathbin{@} t'$.

By the induction hypothesis, $P_1 \xrightarrow{t'} \epsilon$. By Figure 18, $P \xrightarrow{t} \epsilon$.

Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 2.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Case 4.1: $|\sigma(Q)| \leq 1$. The proof is similar to the proof of Case 2.

Case 4.2: $|\sigma(Q)| = r \geq 2$.

Let $x_1, \ldots, x_r$ be the rows of $\sigma(Q)$, $\sigma(Q) = (x_1, \ldots, x_r)$. Let $\sigma_i = \sigma[Q.y \mapsto x_i]$ for each $i = 1, \ldots, r$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, there exists loop layout trees $l_1, \ldots, l_r$ such that $l = (Q, r) \circlearrowleft (l_1, \ldots, l_r)$ and $\sigma_i \vdash P_1 \Downarrow_{\text{loops}} l_i$ for each $i = 1, \ldots, r$. By Algorithm 4, GetAnnotatedTrace(l) = $\cup_{i=1}^{r} \{ \langle Q, r, i \rangle \mathbin{@} t' \mid t' \in$ GetAnnotatedTrace$(l_i) \}$.

Since $t \in$ GetAnnotatedTrace(l), there exists integer $i \in \{1, \ldots, r\}$ and $t' \in$ GetAnnotatedTrace(l_i) such that $t = \langle Q, r, i \rangle \mathbin{@} t'$.

By the induction hypothesis, $P_1 \xrightarrow{t'} \epsilon$. By Figure 18, $P \xrightarrow{t} \epsilon$. □

4.4.2 *Traversing the Loop Layout Tree.*

Proposition 4.36. *For any loop layout trees l_1, l_2, l_3 and annotated traces t_1, t_2:*

(1) *if $l_1 \xhookrightarrow{t_1} l_2$ and $l_2 \xhookrightarrow{t_2} l_3$, then $l_1 \xhookrightarrow{t_1 @ t_2} l_3$.*
(2) *if $l_1 \xhookrightarrow{t_1} l_2$ and $l_1 \xhookrightarrow{t_1 @ t_2} l_3$, then $l_2 \xhookrightarrow{t_2} l_3$.*

Proof. By induction on the length of t_1 and the derivation of l_1. □

Remark. Note that the reverse direction of subtraction does not hold. If $l_2 \xhookrightarrow{t_2} l_3$ and $l_1 \xhookrightarrow{t_1 @ t_2} l_3$, then $l_1 \xhookrightarrow{t_1} l_2$ may not hold. Consider the following loop layout trees:

$$\begin{aligned} l_1 &= (Q_1, 0) \searrow (Q_2, 2) \circlearrowleft (\, (Q_3, 0) \searrow \text{Nil} \,,\ (Q_3, 3) \searrow \text{Nil} \,)\,, \\ l_2 &= (Q_2, 2) \circlearrowleft (\, (Q_3, 0) \searrow \text{Nil} \,,\ (Q_3, 1) \searrow \text{Nil} \,)\,, \\ l_3 &= (Q_3, 0) \searrow \text{Nil}. \end{aligned}$$

Let $t_1 = \langle Q_1, 0, \text{NotLoop} \rangle$ and $t_2 = \langle Q_2, 2, 1 \rangle$. By Figure 19, $l_2 \xhookrightarrow{t_2} l_3$, $l_1 \xhookrightarrow{t_1 @ t_2} l_3$, and $l_1 \xhookrightarrow{t_1} l_2'$ where $l_2' = (Q_2, 2) \circlearrowleft (\, (Q_3, 0) \searrow \text{Nil} \,,\ (Q_3, 3) \searrow \text{Nil} \,)$. However, $l_2 \neq l_2'$.

Proposition 4.37. *For any loop layout trees l_1, l_2 and annotated traces t_1, t_2, if $l_1 \xhookrightarrow{t_1 \,@\, t_2} l_2$, then there exists loop layout tree l_3 such that $l_1 \xhookrightarrow{t_1} l_3$.*

Proof. The proof is by induction on the length of t_1 and the derivation of l_1.

Case 1: $t_1 = \text{Nil}$. Let $l_3 = l_1$. By Figure 19, $l_1 \xhookrightarrow{\text{Nil}} l_1$.
Case 2: $t_1 = \langle Q', r, \lambda\rangle \,@\, t_1'$. We have $t_1 \,@\, t_2 = \langle Q', r, \lambda\rangle \,@\, t_1' \,@\, t_2$.

Case 2.1: $l_1 = \text{Nil}$. Since $t_1 \,@\, t_2 \neq \text{Nil}$, it is not possible to have $l_1 \xhookrightarrow{t_1 \,@\, t_2} l_2$ by Figure 19.
Case 2.2: $l_1 = (Q, r) \searrow l_1'$.

Since $l_1 \xhookrightarrow{t_1 \,@\, t_2} l_2$, by Figure 19, $Q \doteq Q'$, $\lambda = \text{NotLoop}$, and $l_1' \xhookrightarrow{t_1' \,@\, t_2} l_2$. By the induction hypothesis, there exists loop layout tree l_3 such that $l_1' \xhookrightarrow{t_1{}'} l_3$. By Figure 19, $l_1 \xhookrightarrow{\langle Q', r, \lambda\rangle} l_1'$. By Proposition 4.36, $l_1 \xhookrightarrow{t_1} l_3$.

Case 2.3: $l_1 = (Q', r)\circlearrowleft(l_1', \ldots, l_r')$.

Since $l_1 \xhookrightarrow{t_1 \,@\, t_2} l_2$, by Figure 19, $Q \doteq Q'$, $1 \leq \lambda \leq r$, and $l_\lambda' \xhookrightarrow{t_1' \,@\, t_2} l_2$. By the induction hypothesis, there exists loop layout tree l_3 such that $l_\lambda' \xhookrightarrow{t_1{}'} l_3$. By Figure 19, $l_1 \xhookrightarrow{\langle Q', r, \lambda\rangle} l_\lambda'$. By Proposition 4.36, $l_1 \xhookrightarrow{t_1} l_3$. □

Proposition 4.38. *For any loop layout tree l and annotated trace t, we have $t \in$ GetAnnotatedTrace(l) if and only if $l \xhookrightarrow{t}$ Nil.*

Proof. By induction on the length of t. □

4.4.3 *Consistency with Program AST, Path Constraint, and Loop Layout Tree.*

Lemma 4.39. *For any program $P \in Prog$, path constraint W that is derived from P, context $\sigma \in Context$ that satisfies W, if $\sigma \vdash P \Downarrow_{loops} l$, then there exists an annotated trace $t \in$ GetAnnotatedTrace(l) such that $t \sim W$.*

Proof Sketch. The proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

By Figure 17, $l = \text{Nil}$. By Algorithm 4, GetAnnotatedTrace$(l) = \{\text{Nil}\}$.
Let $t = \text{Nil}$, then $t \in$ GetAnnotatedTrace(l).

Case 1.1: $W = \text{Nil}$.

By Definition 3.19, $t \sim W$.

Case 1.2: $W = \langle Q', r', s'\rangle$.

By Definition 4.25, $\mathcal{F}(P) = \text{Nil}$. By Definition 4.4, this case is not possible.

Case 1.3: $W = (\langle Q_1', r_1', s_1'\rangle, \ldots, \langle Q_m', r_m', s_m'\rangle)$, where $m \geq 2$.

By Definition 4.4, there exists an annotated trace t' such that $P \xrightarrow{t'} \epsilon$ and $t' \sim W'$, where $W' = (\langle Q_1', r_1', s_1'\rangle, \ldots, \langle Q_{m-1}', r_{m-1}', s_{m-1}'\rangle, \langle Q_m', r', s_m'\rangle)$ for some row constraint r'. By Figure 18, $t' = \text{Nil}$. By Definition 3.19, this case is not possible.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

Let $\sigma' = \sigma[Q.y \mapsto \sigma(Q)]$. By Figure 19, there exists a loop layout tree l' such that $l = (Q, |\sigma(Q)|) \searrow l'$ and $\sigma' \vdash P_1 \Downarrow_{\text{loops}} l'$.

Case 2.1: $W = \text{Nil}$.

By Algorithm 4, $\textsc{GetAnnotatedTrace}(l') \neq \emptyset$. Hence, there exists an annotated trace $t' \in \textsc{GetAnnotatedTrace}(l')$. Let $t = \langle Q, |\sigma(Q)|, \mathrm{NotLoop}\rangle @ t'$. By Algorithm 4, $t \in \textsc{GetAnnotatedTrace}(l)$. By Definition 3.19, $t \sim W$.

Case 2.2: $W = \langle Q', r', s' \rangle$.

By Definition 3.16, $|\sigma(Q')| \simeq r'$. By Definition 4.25, $\mathcal{F}(P) = Q$. By Definition 4.4, $Q' \doteq_{\mathrm{Nil},\mathrm{Nil},\mathrm{Nil}} Q$. By Definition 3.18, Definition 3.17, and Definition 3.7, $\sigma(Q) = \sigma(Q')$. Hence, $|\sigma(Q)| \simeq r'$.

By Algorithm 4, $\textsc{GetAnnotatedTrace}(l') \neq \emptyset$. Hence, there exists an annotated trace $t' \in \textsc{GetAnnotatedTrace}(l')$. Let $t = \langle Q, |\sigma(Q)|, \mathrm{NotLoop}\rangle @ t'$. By Algorithm 4, $t \in \textsc{GetAnnotatedTrace}(l)$. By Definition 3.19, $t \sim W$.

Case 2.3: $W = (\langle Q_1', r_1', s_1' \rangle, \ldots, \langle Q_m', r_m', s_m' \rangle)$, where $m \geq 2$.

By Definition 4.4, there exists an annotated trace t' such that $P \xrightarrow{t'} \epsilon$ and $t' \sim W'$, where $W' = (\langle Q_1', r_1', s_1' \rangle, \ldots, \langle Q_{m-1}', r_{m-1}', s_{m-1}' \rangle, \langle Q_m', r', s_m' \rangle)$ for some row constraint r'.

Let $t' = \langle Q'', r'', \lambda'' \rangle @ t''$. By Definition 3.19, $Q'' \doteq_{\mathrm{Nil},\mathrm{Nil},\mathrm{Nil}} Q_1'$. Also, $r'' \simeq r_1'$. By Figure 18, $Q'' \doteq_{\mathrm{Nil},\mathrm{Nil},\mathrm{Nil}} Q$.

Hence, $Q_1' \doteq_{\mathrm{Nil},\mathrm{Nil},\mathrm{Nil}} Q$. By Definition 3.18, Definition 3.17, and Definition 3.7, $\sigma(Q) = \sigma(Q_1')$.

Since σ satisfies W, by Definition 3.16, there exists a sequence of contexts $\sigma_1, \ldots, \sigma_m \in \mathit{Context}$ that are updated according to the evaluation of the queries $Q_1', \ldots, Q_m'$ in σ and $|\sigma_i(Q_i')| \simeq r_i'$ for all $i = 1, \ldots, m$. Since $\sigma_1 = \sigma$, we have $\sigma(Q) = \sigma_1(Q_1')$. Hence, $|\sigma(Q)| \simeq r_1'$.

Since $P \xrightarrow{t'} \epsilon$, by Figure 18, $P_1 \xrightarrow{t''} \epsilon$.

Let $W'' = (\langle Q_2', r_2', s_2' \rangle, \ldots, \langle Q_{m-1}', r_{m-1}', s_{m-1}' \rangle, \langle Q_m', r', s_m' \rangle)$. By Definition 3.19, $t'' \sim W''$.

Let $W''' = (\langle Q_2', r_2', s_2' \rangle, \ldots, \langle Q_m', r_m', s_m' \rangle)$. By Definition 4.4, W''' is derived from P_1. By Definition 3.16, σ_2 satisfies W'''. Also, $\sigma_2 = \sigma_1[Q_1'.y \mapsto \sigma_1(Q_1')] = \sigma[Q.y \mapsto \sigma(Q)] = \sigma'$. Hence, σ' satisfies W'''.

Since $\sigma' \vdash P_1 \Downarrow_{\mathrm{loops}} l'$, by the induction hypothesis, there exists an annotated trace $t''' \in \textsc{GetAnnotatedTrace}(l')$ such that $t''' \sim W'''$.

Let $t = \langle Q, |\sigma(Q)|, \mathrm{NotLoop}\rangle @ t'''$. Since $l = (Q, |\sigma(Q)|) \searrow l'$, by Algorithm 4, $t \in \textsc{GetAnnotatedTrace}(l)$.

Since $|\sigma(Q)| \simeq r_1'$, by Definition 3.19, $t \sim W$.

Case 3: P is of the form “If”. P expands to “`if` Q, `then` P_1 `else` P_2”, where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Case 3.1: $|\sigma(Q)| \geq 1$.

Let $\sigma' = \sigma[Q.y \mapsto \sigma(Q)]$. By Figure 19, there exists a loop layout tree l' such that $l = (Q, |\sigma(Q)|) \searrow l'$ and $\sigma' \vdash P_1 \Downarrow_{\mathrm{loops}} l'$.

Case 3.1.1: $W = \mathrm{Nil}$. The proof is similar to the proof of Case 2.1.

Case 3.1.2: $W = \langle Q', r', s' \rangle$. The proof is similar to the proof of Case 2.2.

Case 3.1.3: $W = (\langle Q_1', r_1', s_1' \rangle, \ldots, \langle Q_m', r_m', s_m' \rangle)$, where $m \geq 2$.

The proof is similar to the proof of Case 2.3. The main modification is the proof of $P_1 \xrightarrow{t''} \epsilon$: Since $|\sigma(Q)| \geq 1$, by Definition 3.15, $r_1' = (\geq 1)$ or $r_1' = (\geq 2)$. Either case, since $r'' \simeq r_1'$, we have $r'' \geq 1$. Since $P \xrightarrow{t'} \epsilon$, by Figure 18, $P_1 \xrightarrow{t''} \epsilon$.

Case 3.2: $|\sigma(Q)| = 0$. The proof is similar to the proof of Case 3.1.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Case 4.1: $|\sigma(Q)| \geq 2$.

Let $\sigma(Q) = (x_1, \ldots, x_r)$, where $r = |\sigma(Q)| \geq 2$. Let $\sigma'_i = \sigma[Q.y \mapsto x_i]$ for each $i = 1, \ldots, r$. By Figure 17, there exists loop layout trees $l'_1, \ldots, l'_r$ such that $l = (Q, r)\circlearrowleft(l'_1, \ldots, l'_r)$ and $\sigma'_i \vdash P_1 \Downarrow_{\text{loops}} l'_i$ for all $i = 1, \ldots, r$.

Case 4.1.1: $W = \text{Nil}$. The proof is similar to the proof of Case 2.1.

Case 4.1.2: $W = \langle Q', r', s' \rangle$. The proof is similar to the proof of Case 2.2.

Case 4.1.3: $W = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$, where $m \geq 2$.

The proof is similar to the proof of Case 3.1.3. The main modifications are the reasoning after defining W''':

By Definition 4.4, $s'_1 = \texttt{true}$. By Definition 3.16, there exists integer k_1 such that $1 \leq k_1 \leq |\sigma_1(Q'_1)| = |\sigma(Q)| = r$ and $\sigma_2 = \sigma_1[Q'_1.y \mapsto \sigma_1(Q'_1)[k_1]] = \sigma[Q.y \mapsto x_{k_1}] = \sigma'_{k_1}$. Hence, σ'_{k_1} satisfies W'''.

Since $\sigma'_{k_1} \vdash P_1 \Downarrow_{\text{loops}} l'_{k_1}$, by the induction hypothesis, there exists an annotated trace $t''' \in \textsc{GetAnnotatedTrace}(l'_{k_1})$ such that $t''' \sim W'''$.

Let $t = \langle Q, r, k_1 \rangle \mathbin{@} t'''$. Since $l = (Q, r)\circlearrowleft(l'_1, \ldots, l'_r)$, by Algorithm 4, $t \in \textsc{GetAnnotatedTrace}(l)$.

Since $r = |\sigma(Q)| \simeq r'_1$, by Definition 3.19, $t \sim W$.

Case 4.2: $|\sigma(Q)| = 1$. The proof is similar to the proof of Case 3.1.

Case 4.3: $|\sigma(Q)| = 0$. The proof is similar to the proof of Case 3.1.

In this proof sketch, we reuse the notation in Definition 3.19 when stating "$t'' \sim W''$" in Case 2.3. To complete the proof, we slightly revise this expression, as well as the expression "$t \sim W$" in the induction hypothesis, as follows: Generalize Definition 3.19 to work with subprograms. Specifically, define what it means for a suffix of an annotated trace to be consistent with a suffix of a path constraint, with respect to a prefix of the path constraint. This prefix of the path constraint specifies the path through the program to reach the subprogram that generates the trace suffix. Passing along this prefix of the path constraint is straightforward, as we have done systematically in Figure 15, Algorithm 8, and Lemma 4.21. This prefix of the path constraint is useful for reasoning about the equivalence of the queries in t'' and W''. □

Lemma 4.40. *For any program $P \in Prog$, path constraint W that is derived from P, context $\sigma \in Context$ that satisfies W, if $\sigma \vdash P \Downarrow_{loops} l$ and $l \neq Nil$, then $\textsc{MatchPath}(l, W) \neq Nil$.*

Proof.

Case 1: $W = \text{Nil}$.

Since $l \neq \text{Nil}$, by Algorithm 4, there exists an annotated trace $t \in \textsc{GetAnnotatedTrace}(l)$ such that $t \neq \text{Nil}$. By Definition 3.19, $t \sim \text{Nil}$. In Algorithm 4, execution enters the branch on line 6 with $t \neq \text{Nil}$.

Case 2: $W = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$ and $m \geq 1$.

By Lemma 4.39, there exists an annotated trace $t' \in \textsc{GetAnnotatedTrace}(l)$ such that $t' \sim W$. Since $m \geq 1$, by Definition 3.19, $t' \neq \text{Nil}$. In Algorithm 4, execution eventually enters the branch on line 6 with variable $t \neq \text{Nil}$. □

Lemma 4.41. *For any program $P \in Prog$, path constraint W that is derived from P, context $\sigma \in Context$ that satisfies W, loop layout tree l such that $\sigma \vdash P \Downarrow_{loops} l$, and annotated trace $t = \textsc{MatchPath}(l, W)$:*

(1) $t \sim W$.
(2) $P \xrightarrow{t} \epsilon$.
(3) $l \xhookrightarrow{t} Nil$.

Proof.

Case 1: $l = \text{Nil}$.

By Figure 17, $P = \epsilon$. By Algorithm 4, $\text{GetAnnotatedTrace}(l) = \{\,\text{Nil}\,\}$. In Algorithm 4, execution never enters line 6 and thus returns Nil on line 10. Hence, $t = \text{MatchPath}(l, W) = \text{Nil}$. By Figure 18, $P \xrightarrow{t} \epsilon$. By Figure 19, $l \xhookrightarrow{t} \text{Nil}$.

By Definition 4.4, $W = \text{Nil}$. By Definition 3.19, $t \sim W$.

Case 2: $l \neq \text{Nil}$.

By Lemma 4.40, $t = \text{MatchPath}(l, W) \neq \text{Nil}$. In Algorithm 4, execution must not return on line 10. Since $\text{GetAnnotatedTrace}(l)$ contains a finite number of annotated traces, the execution must return on line 7. Hence, $t \in \text{GetAnnotatedTrace}(l)$ and $t \sim W$.

By Proposition 4.35, $P \xrightarrow{t} \epsilon$. By Proposition 4.38, $l \xhookrightarrow{t} \text{Nil}$. □

Theorem 2 (Trace-Code Correspondence). *For any program $P \in \mathcal{K}$, path constraint W that is derived from P, context $\sigma \in Context$ that satisfies W, and annotated trace t, if $t = \text{GetTrace}(\boxed{P}, W, \sigma)$, then there exists a loop layout tree l' such that:*

(1) $\sigma \vdash P \Downarrow_{loops} l'$,
(2) $t \sim W$,
(3) $P \xrightarrow{t} \epsilon$,
(4) $l' \xhookrightarrow{t} Nil$, *and*
(5) *l' and the variable l are identical except for equivalent variables.*

Proof. Let e' be a list of query-result pairs such that $\sigma \vdash P \Downarrow_{\text{exec}} e'$. By Proposition 4.7, the variable e in Algorithm 2 and e' are identical except for equivalent variables.

Let $l' = \text{DetectLoops}(e')$. Since the variable $l = \text{DetectLoops}(e)$, l and l' are also identical except for equivalent variables. By Theorem 1, $\sigma \vdash P \Downarrow_{\text{loops}} l'$.

Let $t' = \text{MatchPath}(l', W)$. Since the variable $t = \text{MatchPath}(l, W)$, t and t' are also identical except for equivalent variables. By Lemma 4.41, $t' \sim W$, $P \xrightarrow{t'} \epsilon$, and $l' \xhookrightarrow{t'} \text{Nil}$.

By Figure 18, $P \xrightarrow{t} \epsilon$. By Definition 3.19, $t \sim W$. By Figure 19, $l' \xhookrightarrow{t} \text{Nil}$. □

4.5 Soundness of the Core Inference Algorithm

To help characterize the execution of the core inference algorithm InferProg, we first present a notation for reasoning about context updates (Section 4.5.1). We then present the soundness proof of InferProg in Section 4.5.2. We conclude with the soundness proof of Infer in Section 4.5.3.

4.5.1 *Updating the Context while Traversing the Program AST.*

Figure 20 presents the definition of simultaneously updating the context, traversing a program $P \in \text{Prog}$, and traversing a loop layout tree, by following an annotated trace.

Proposition 4.42. *For any programs $P_1, P_2, P_3 \in Prog$, contexts $\sigma_1, \sigma_2, \sigma_3 \in Context$, loop layout trees l_1, l_2, l_3, and annotated traces t_1, t_2:*

(1) $\text{if} \begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix} \xrightarrow{t_1}\!\!\!\!\rightarrow \begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix} \text{ and } \begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix} \xrightarrow{t_2}\!\!\!\!\rightarrow \begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix} \text{ then } \begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix} \xrightarrow{t_1\,@\,t_2}\!\!\!\!\rightarrow \begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}.$

$$\frac{}{\begin{bmatrix}\sigma\\P\\l\end{bmatrix}\overset{\text{Nil}}{\twoheadrightarrow}\begin{bmatrix}\sigma\\P\\l\end{bmatrix}}\quad\text{(nil)}$$

$$\frac{|\sigma(Q)|=r\qquad\begin{bmatrix}\sigma[Q.y\mapsto\sigma(Q)]\\P\\l\end{bmatrix}\overset{t}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}\qquad Q\doteq Q'}{\begin{bmatrix}\sigma\\Q\ P\\(Q,r)\searrow l\end{bmatrix}\xrightarrow{\langle Q',r,\text{NotLoop}\rangle\,@\,t}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}}\quad\text{(seq)}$$

$$\frac{|\sigma(Q)|=r>0\qquad\begin{bmatrix}\sigma[Q.y\mapsto\sigma(Q)]\\P_1\\l\end{bmatrix}\overset{t}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P_1'\\l'\end{bmatrix}\qquad Q\doteq Q'}{\begin{bmatrix}\sigma\\\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2\\(Q,r)\searrow l\end{bmatrix}\xrightarrow{\langle Q',r,\text{NotLoop}\rangle\,@\,t}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma'\\P_1'\\l'\end{bmatrix}}\quad\text{(if-1)}$$

$$\frac{|\sigma(Q)|=0\qquad\begin{bmatrix}\sigma\\P_2\\t\end{bmatrix}\overset{l}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P_2'\\l'\end{bmatrix}\qquad Q\doteq Q'}{\begin{bmatrix}\sigma\\\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2\\(Q,0)\searrow l\end{bmatrix}\xrightarrow{\langle Q',0,\text{NotLoop}\rangle\,@\,t}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma'\\P_2'\\l'\end{bmatrix}}\quad\text{(if-2)}$$

$$\frac{\sigma(Q)=(x_1,\ldots,x_r)\qquad r\geq 2\qquad\begin{bmatrix}\sigma[Q.y\mapsto x_i]\\P_1\\l_i\end{bmatrix}\overset{t}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P_1'\\l'\end{bmatrix}\qquad Q\doteq Q'}{\begin{bmatrix}\sigma\\\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2\\(Q,r)\circlearrowleft(l_1,\ldots,l_r)\end{bmatrix}\xrightarrow{\langle Q',r,i\rangle\,@\,t}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma'\\P_1'\\l'\end{bmatrix}}\quad\text{(for-1a)}$$

$$\frac{|\sigma(Q)|=1\qquad\begin{bmatrix}\sigma[Q.y\mapsto\sigma(Q)]\\P_1\\l\end{bmatrix}\overset{t}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P_1'\\l'\end{bmatrix}\qquad Q\doteq Q'}{\begin{bmatrix}\sigma\\\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2\\(Q,1)\searrow l\end{bmatrix}\xrightarrow{\langle Q',1,\text{NotLoop}\rangle\,@\,t}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma'\\P_1'\\l'\end{bmatrix}}\quad\text{(for-1b)}$$

$$\frac{|\sigma(Q)|=0\qquad\begin{bmatrix}\sigma\\P_2\\l\end{bmatrix}\overset{t}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P_2'\\l'\end{bmatrix}\qquad Q\doteq Q'}{\begin{bmatrix}\sigma\\\texttt{for}\ Q\ \texttt{for}\ P_1\ \texttt{else}\ P_2\\(Q,0)\searrow l\end{bmatrix}\xrightarrow{\langle Q',0,\text{NotLoop}\rangle\,@\,t}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma'\\P_2'\\l'\end{bmatrix}}\quad\text{(for-2)}$$

$$P,P_1,P_2,P',P_1',P_2'\in\text{Prog},\qquad Q,Q'\in\text{Query},\qquad r,i\in\mathbb{Z}_{\geq 0}$$

Fig. 20. Traverse a program and a corresponding loop layout tree by following an annotated trace, updating the context.

$$\textit{(2) if}\begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix}\overset{t_1}{\twoheadrightarrow}\begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix}\textit{ and }\begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix}\xrightarrow{t_1\,@\,t_2}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}\textit{ then }\begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix}\overset{t_2}{\twoheadrightarrow}\begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}.$$

Proof. By induction on the length of t_1 and the derivation of P_1. □

Remark. Note that the reverse direction of subtraction does not hold. If $\begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix}\overset{t_2}{\twoheadrightarrow}\begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}$ and $\begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix}\xrightarrow{t_1\,@\,t_2}\!\!\!\!\!\twoheadrightarrow\begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}$, $\begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix}\overset{t_1}{\twoheadrightarrow}\begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix}$ may not hold. Counter examples are similar to that of Sections 4.4.1 and 4.4.2.

Proposition 4.43. *For any programs $P,P'\in$ Prog, contexts $\sigma,\sigma'\in$ Context, loop layout trees l,l', annotated trace t, if $\begin{bmatrix}\sigma\\P\\l\end{bmatrix}\overset{t}{\twoheadrightarrow}\begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}$, then $P\overset{t}{\to}P'$ and $l\overset{t}{\hookrightarrow}l'$.*

Proof. By induction on the derivation of P. □

Proposition 4.44. *For any programs $P, P' \in Prog$, context $\sigma \in Context$, loop layout trees l, l', and annotated query tuple $\langle Q', r, \lambda \rangle$:*

(1) *if $\sigma \vdash P \Downarrow_{loops} l$, $P \xrightarrow{\langle Q', r, \lambda \rangle} P'$, and $l \xhookrightarrow{\langle Q', r, \lambda \rangle} l'$, then there exists $\sigma' \in Context$ such that $\overset{\sigma}{\underset{l}{[P]}} \xrightarrow{\langle Q', r, \lambda \rangle} \overset{\sigma'}{\underset{l'}{[P']}}$.*

(2) *for any context $\sigma' \in Context$, if $\sigma \vdash P \Downarrow_{loops} l$ and $\overset{\sigma}{\underset{l}{[P]}} \xrightarrow{\langle Q', r, \lambda \rangle} \overset{\sigma'}{\underset{l'}{[P']}}$, then $\sigma' \vdash P' \Downarrow_{loops} l'$.*

Proof.

(1) By induction on the derivation of P.
(2) This proof is by induction on the derivation of P.
Case 1: $P = \epsilon$.
By Figure 20, it is not possible to have $\overset{\sigma}{\underset{l}{[P]}} \xrightarrow{\langle Q', r, \lambda \rangle} \overset{\sigma'}{\underset{l'}{[P']}}$.
Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.
By Figure 20, $r = |\sigma(Q)|$, $\sigma' = \sigma[Q.y \mapsto \sigma(Q)]$, $P' = P_1$, and $l = (Q, r) \searrow l'$.
Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.
Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof for Case 2.
Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.
Case 4.1: $|\sigma(Q)| \leq 1$. The proof is similar to the proof for Case 2.
Case 4.2: $|\sigma(Q)| \geq 2$.
Let $(x_1, \ldots, x_r) = \sigma(Q)$.
By Figure 20, $r = |\sigma(Q)| \geq 2$, $\lambda \in \{1, \ldots, r\}$, $\sigma' = \sigma[Q.y \mapsto x_\lambda]$, $P' = P_1$, and $l' = l_\lambda$. Also, there exists $l_1, \ldots, l_r$ such that $l = (Q, r) \circlearrowright (l_1, \ldots, l_r)$.
Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 17, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$. □

Proposition 4.45. *For any programs $P, P' \in Prog$, context $\sigma \in Context$, loop layout trees l, l', and annotated trace t:*

(1) *if $\sigma \vdash P \Downarrow_{loops} l$, $P \xrightarrow{t} P'$, and $l \xhookrightarrow{t} l'$, then there exists $\sigma' \in Context$ such that $\overset{\sigma}{\underset{l}{[P]}} \xrightarrow{t} \overset{\sigma'}{\underset{l'}{[P']}}$.*

(2) *for any context $\sigma' \in Context$, if $\sigma \vdash P \Downarrow_{loops} l$ and $\overset{\sigma}{\underset{l}{[P]}} \xrightarrow{t} \overset{\sigma'}{\underset{l'}{[P']}}$, then $\sigma' \vdash P' \Downarrow_{loops} l'$.*

Proof.

(1) This proof is by induction on the length of t.
Case 1: $t = \text{Nil}$.
By Figure 18, $P' = P$. By Figure 19, $l' = l$. By Figure 20, $\overset{\sigma}{\underset{l}{[P]}} \xrightarrow{\text{Nil}} \overset{\sigma}{\underset{l}{[P]}}$.
Case 2: $t = \langle Q', r, \lambda \rangle @ t''$.

By Proposition 4.34, there exists $P'' \in \mathrm{Prog}$ such that $P \xrightarrow{\langle Q',r,\lambda\rangle} P''$. By Proposition 4.37, there exists l'' such that $l \xhookrightarrow{\langle Q',r,\lambda\rangle} l''$. By Proposition 4.44, there exists $\sigma'' \in \mathit{Context}$ such that $[\overset{\sigma}{\underset{l}{P}}] \overset{\langle Q',r,\lambda\rangle}{\twoheadrightarrow} [\overset{\sigma''}{\underset{l''}{P''}}]$. By Proposition 4.44, $\sigma'' \vdash P'' \Downarrow_{\text{loops}} l''$.
Since $P \xrightarrow{t} P'$, we have $P \xrightarrow{\langle Q',r,\lambda\rangle @ t''} P'$. By Proposition 4.33, $P'' \xrightarrow{t''} P'$.
Since $l \xhookrightarrow{t} l'$, we have $l \xhookrightarrow{\langle Q',r,\lambda\rangle @ t''} l'$. By Proposition 4.36, $l'' \xhookrightarrow{t''} l'$.
By the induction hypothesis, there exists $\sigma''' \in \mathit{Context}$ such that $[\overset{\sigma''}{\underset{l''}{P''}}] \overset{t''}{\twoheadrightarrow} [\overset{\sigma'''}{\underset{l'}{P'}}]$.
Since $[\overset{\sigma}{\underset{l}{P}}] \overset{\langle Q',r,\lambda\rangle}{\twoheadrightarrow} [\overset{\sigma''}{\underset{l''}{P''}}]$, by Proposition 4.42, $[\overset{\sigma}{\underset{l}{P}}] \overset{\langle Q',r,\lambda\rangle @ t''}{\twoheadrightarrow} [\overset{\sigma'''}{\underset{l'}{P'}}]$. Hence, $[\overset{\sigma}{\underset{l}{P}}] \overset{t}{\twoheadrightarrow} [\overset{\sigma'''}{\underset{l'}{P'}}]$.

(2) This proof is by induction on the length of t.
<u>Case 1:</u> $t = \mathrm{Nil}$.
By Figure 20, $\sigma' = \sigma$, $P' = P$, and $l' = l$. Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.
<u>Case 2:</u> $t = \langle Q',r,\lambda\rangle @ t''$.
By Proposition 4.43, $P \xrightarrow{t} P'$ and $l \xhookrightarrow{t} l'$. By Proposition 4.34, there exists $P'' \in \mathrm{Prog}$ such that $P \xrightarrow{\langle Q',r,\lambda\rangle} P''$. By Proposition 4.37, there exists l'' such that $l \xhookrightarrow{\langle Q',r,\lambda\rangle} l''$.
Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Proposition 4.44, there exists σ'' such that $[\overset{\sigma}{\underset{l}{P}}] \overset{\langle Q',r,\lambda\rangle}{\twoheadrightarrow} [\overset{\sigma''}{\underset{l''}{P''}}]$. By Proposition 4.44, $\sigma'' \vdash P'' \Downarrow_{\text{loops}} l''$.
Since $[\overset{\sigma}{\underset{l}{P}}] \overset{t}{\twoheadrightarrow} [\overset{\sigma'}{\underset{l'}{P'}}]$, by Proposition 4.42, $[\overset{\sigma''}{\underset{l''}{P''}}] \overset{t''}{\twoheadrightarrow} [\overset{\sigma'}{\underset{l'}{P'}}]$.
By the induction hypothesis, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$. ☐

4.5.2 Soundness of InferProg. To facilitate discussion, we define an alternative implementation of InferProg in Algorithm 10. This version is equivalent to Algorithm 6 and uses annotated traces more explicitly. We first present a detailed case-by-case discussion on the properties of the variables in Algorithm 10 by line 16. We then conclude with the proof of Theorem 3.

Proposition 4.46. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$ and annotated traces t, t' such that $t \neq Nil$, $t' \neq Nil$, $P' \xrightarrow{t'} P$, and $P \xrightarrow{t} \epsilon$. During the execution of* InferProg($\boxed{P'}$, t', t)*, for each $i = 0, 1, 2$, the variable W_i on line 9 of Algorithm 10 is derived from P'.*

Proof. By Proposition 4.33, $P' \xrightarrow{t' @ t} \epsilon$. By Definition 3.19, Definition 4.4, and the definition of MakePathConstraint, W_i is derived from P'. ☐

Lemma 4.47. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$ and annotated traces t, t' such that $t \neq Nil$, $t' \neq Nil$, $P' \xrightarrow{t'} P$, and $P \xrightarrow{t} \epsilon$. During the execution of* InferProg($\boxed{P'}$, t', t)*, let σ_i' be the context variable in* SolveAndGetTrace *on line 10 of Algorithm 10 for each integer $i \in \{0,1,2\}$. If variable $f_i =$* `true` *on line 16, then there exists $\sigma_i, \sigma_i'', P'', l_i, l_i', l_i''$ such that $[\overset{\sigma_i'}{\underset{l_i'}{P'}}] \overset{\langle Q_1, r_{i,1}, \lambda_{i,1}\rangle, \ldots, \langle Q_k, r_{i,k}, \lambda_{i,k}\rangle}{\twoheadrightarrow} [\overset{\sigma_i}{\underset{l_i}{P}}]$ and $[\overset{\sigma_i}{\underset{l_i}{P}}] \overset{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1}\rangle}{\twoheadrightarrow} [\overset{\sigma_i''}{\underset{l_i''}{P''}}]$.*

ALGORITHM 10: Recursively infer a subprogram (Algorithm 6) with more detail

Input: $\boxed{P}$ is the executable of a program $P \in \mathcal{K}$.
Input: s_1 is a prefix of an annotated trace.
Input: s_2 is a suffix of an annotated trace.
Output: Subprogram equivalent to P's subprogram after trace s_1.

procedure InferProg($\boxed{P}, s_1, s_2$)
  if $s_2 =$ Nil **then return** ϵ ▷ Prog := ϵ
  end if
  $k \leftarrow$ The length of s_1
  if $k > 0$ **then** $\langle Q_1, r_1, \lambda_1 \rangle, \ldots, \langle Q_k, r_k, \lambda_k \rangle \leftarrow s_1$
  end if
  $\langle Q_{k+1}, r_{k+1}, \lambda_{k+1} \rangle, \ldots, \langle Q_n, r_n, \lambda_n \rangle \leftarrow s_2$
  for $i = 0, 1, 2$ **do**
    $W_i \leftarrow$ MakePathConstraint(s_1, Q_{k+1}, i)
    $(f_i, t_i) \leftarrow$ SolveAndGetTrace($\boxed{P}, W_i$)
    if f_i **then** ▷ Satisfiable
      $\langle Q_1, r_{i,1}, \lambda_{i,1} \rangle, \ldots, \langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle,$
        $\langle Q_{i,k+2}, r_{i,k+2}, \lambda_{i,k+2} \rangle, \ldots, \langle Q_{i,m_i}, r_{i,m_i}, \lambda_{i,m_i} \rangle \leftarrow t_i$
      $t_{i,1} \leftarrow \langle Q_1, r_{i,1}, \lambda_{i,1} \rangle, \ldots, \langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle$ ▷ New trace prefix
      $t_{i,2} \leftarrow \langle Q_{i,k+2}, r_{i,k+2}, \lambda_{i,k+2} \rangle, \ldots, \langle Q_{i,m_i}, r_{i,m_i}, \lambda_{i,m_i} \rangle$ ▷ New trace suffix
    end if
  end for
  if f_2 **and** $\lambda_{2,k+1} \neq$ NotLoop **then**
    $b_t \leftarrow$ InferProg($\boxed{P}, t_{2,1}, t_{2,2}$)
    if f_0 **then** $b_f \leftarrow$ InferProg($\boxed{P}, t_{0,1}, t_{0,2}$)
    else $b_f \leftarrow \epsilon$
    end if
    return "`for` Q_{k+1} `do` b_t `else` b_f" ▷ Prog := For
  else if f_0 **and** f_1 **and** $((t_{0,2} =$ Nil **and** $t_{1,2} \neq$ Nil$)$ **or** $(t_{0,2} \neq$ Nil **and** $t_{1,2} =$ Nil$)$ **or**
      $(t_{0,2} \neq$ Nil **and** $t_{1,2} \neq$ Nil **and** $\pi_{\mathbb{S}} Q_{0,k+2} \neq \pi_{\mathbb{S}} Q_{1,k+2}))$ **then**
    $b_t \leftarrow$ InferProg($\boxed{P}, t_{1,1}, t_{1,2}$)
    $b_f \leftarrow$ InferProg($\boxed{P}, t_{0,1}, t_{0,2}$)
    return "`if` Q_{k+1} `then` b_t `else` b_f" ▷ Prog := If
  else
    if f_0 **then** $b \leftarrow$ InferProg($\boxed{P}, t_{0,1}, t_{0,2}$)
    else $b \leftarrow$ InferProg($\boxed{P}, t_{1,1}, t_{1,2}$)
    end if
    return "Q_{k+1} b" ▷ Prog := Seq
  end if
end procedure

Proof. In Algorithm 10, variables $s_1 = t'$ and $s_2 = t$. By Proposition 4.46, W_i is derived from P'.
Since $f_i =$ `true`, by Algorithm 5, variable $t_i =$ GetTrace($\boxed{P'}, W_i, \sigma_i'$). By Theorem 2, there exists a loop layout tree l_i' such that:

$$\sigma_i' \vdash P' \Downarrow_{\text{loops}} l_i', \tag{1}$$

$$t_i \sim W_i, \tag{2}$$

$$P' \xrightarrow{t_i} \epsilon, \tag{3}$$

$$l'_i \xhookrightarrow{t_i} \text{Nil}. \tag{4}$$

Since $f_i = \texttt{true}$, variables $t_{i,1}$ and $t_{i,2}$ are defined on line 16 and satisfy:

$$t_i = t_{i,1} \,@\, t_{i,2}. \tag{5}$$

Since $t' \neq \text{Nil}$, variable $k \geq 1$ on line 6. Let $t'_{i,1} = \langle Q_1, r_{i,1}, \lambda_{i,1}\rangle, \ldots, \langle Q_k, r_{i,k}, \lambda_{i,k}\rangle$, then:

$$t_{i,1} = t'_{i,1} \,@\, \langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1}\rangle. \tag{6}$$

By Equations (4), (5), (6), and Proposition 4.37, there exists l_i, l''_i such that:

$$l'_i \xhookrightarrow{t'_{i,1}} l_i, \tag{7}$$

$$l_i \xhookrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1}\rangle} l''_i. \tag{8}$$

For all $j = 1, \ldots, k$, by Equation (2), variable $r_{i,j} = 0$ if and only if variable $r_j = 0$. Hence, traversing a program by following t' or $t'_{i,1}$ will use the same rules in Figure 18. Since $P' \xrightarrow{t'} P$,

$$P' \xrightarrow{t'_{i,1}} P. \tag{9}$$

By Equations (1), (9), (7), and Proposition 4.45, there exists σ_i such that:

$$\begin{bmatrix} \sigma'_i \\ P' \\ l'_i \end{bmatrix} \xrightarrow{t'_{i,1}}\!\!\!\!\rightarrow \begin{bmatrix} \sigma_i \\ P \\ l_i \end{bmatrix}. \tag{10}$$

By Equations (3), (5), (6), (9), and Proposition 4.33,

$$P \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1}\rangle \,@\, t_{i,2}} \epsilon. \tag{11}$$

By Equation (11) and Proposition 4.34, there exists P'' such that:

$$P \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1}\rangle} P''. \tag{12}$$

By Equations (1), (10), and Proposition 4.45,

$$\sigma_i \vdash P \Downarrow_{\text{loops}} l_i. \tag{13}$$

By Equations (13), (12), (8), and Proposition 4.45, there exists σ''_i such that $[\begin{smallmatrix} \sigma_i \\ P \\ l_i \end{smallmatrix}] \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1}\rangle}\!\!\!\!\rightarrow [\begin{smallmatrix} \sigma''_i \\ P'' \\ l''_i \end{smallmatrix}]$. □

Lemma 4.48. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$ and annotated traces t, t' such that $t \neq Nil$, $P' \xrightarrow{t'} P$, and $P \xrightarrow{t} \epsilon$. During the execution of* InferProg($\boxed{P'}$, t', t)*, if variable $f_2 = \texttt{true}$ on line 16, then variable $\lambda_{2,k+1} \neq NotLoop$ if and only if P is of the form "For".*

Proof. In Algorithm 10, variables $s_1 = t'$ and $s_2 = t$. By Proposition 4.46, W_2 is derived from P'.

Let σ'_2 be the context variable in SolveAndGetTrace on line 10 for $i = 2$. Since $f_2 = \texttt{true}$, by Algorithm 5, variable $t_2 = $ GetTrace($\boxed{P'}$, W_2, σ'_2). By Theorem 2, there exists a loop layout tree l'_2 such that $\sigma'_2 \vdash P' \Downarrow_{\text{loops}} l'_2$, $t_2 \sim W_2$, and $l'_2 \xhookrightarrow{t_2} \text{Nil}$.

Case 1: $t' = \text{Nil}$.

Since $P' \xrightarrow{t'} P$, by Figure 18, $P' = P$.

Let $\sigma_2 = \sigma_2'$, $l_2 = l_2'$, then $\sigma_2 \vdash P \Downarrow_{\text{loops}} l_2$.

Since $t' = \text{Nil}$, variable $k = 0$ on line 6. Variable $t_{2,1} = \langle Q_1, r_{2,1}, \lambda_{2,1}\rangle$ on line 16. Variable $t_2 = t_{2,1} @ t_{2,2} = \langle Q_1, r_{2,1}, \lambda_{2,1}\rangle @ t_{2,2}$.

Since $l_2' \xhookrightarrow{t_2} \text{Nil}$, by Proposition 4.37, there exists l_2'' such that $l_2' \xhookrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1}\rangle} l_2''$. Hence, $l_2 \xhookrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1}\rangle} l_2''$.

By the definition of MakePathConstraint, the first query in W_2 is the first query in t. By Definition 3.19, the first queries in t_2 and W_2 are identical except for equivalent variables. In other words, Q_1 and the first query in t are identical except for equivalent variables. Since $P \xrightarrow{t} \epsilon$ and $t \neq \text{Nil}$, by Figure 18, there exists P'' such that $P \xrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1}\rangle} P''$.

By Proposition 4.44, there exists σ'' such that $[\overset{\sigma_2}{\underset{l_2}{P}}] \xrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1}\rangle}\!\!\!\!\rightarrow [\overset{\sigma_2''}{\underset{l_2''}{P''}}]$.

Case 2: $t' \neq \text{Nil}$.

By Lemma 4.47, there exists $\sigma_2, \sigma_2'', P'', l_2, l_2''$ such that $[\overset{\sigma_2'}{\underset{l_2'}{P'}}] \xrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1}\rangle, \ldots, \langle Q_k, r_{2,k}, \lambda_{2,k}\rangle}\!\!\!\!\rightarrow [\overset{\sigma_2}{\underset{l_2}{P}}]$ and $[\overset{\sigma_2}{\underset{l_2}{P}}] \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1}\rangle}\!\!\!\!\rightarrow [\overset{\sigma_2''}{\underset{l_2''}{P''}}]$.

Either case, we have $[\overset{\sigma_2}{\underset{l_2}{P}}] \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1}\rangle}\!\!\!\!\rightarrow [\overset{\sigma_2''}{\underset{l_2''}{P''}}]$.

The rest of the proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

Since $P \xrightarrow{t} \epsilon$, by Figure 18, it is not possible to have $t \neq \text{Nil}$. Hence, the proposition trivially holds.

Case 2: P is of the form "Seq". P expands to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

Since $[\overset{\sigma_2}{\underset{l_2}{P}}] \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1}\rangle}\!\!\!\!\rightarrow [\overset{\sigma_2''}{\underset{l_2''}{P''}}]$, by Figure 20, $\lambda_{2,k+1} = \text{NotLoop}$.

Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. The proof is similar to the proof of Case 2.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Since $t_2 \sim W_2$, by Definition 3.19 and the definition of MakePathConstraint, $r_{2,k+1} \geq 2$. Since $[\overset{\sigma_2}{\underset{l_2}{P}}] \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1}\rangle}\!\!\!\!\rightarrow [\overset{\sigma_2''}{\underset{l_2''}{P''}}]$, by Figure 20, $\lambda_{2,k+1} \neq \text{NotLoop}$. □

Lemma 4.49. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$, where P is of the form "Seq", and any annotated traces t, t' such that $P' \xrightarrow{t'} P$ and $P \xrightarrow{t} \epsilon$. Let P expand to "$Q\ P_1$" where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol. During the execution of InferProg($\boxed{P'}$, t', t), for any integer $i \in \{0, 1, 2\}$, if variable f_i = `true` on line 16, then $P' \xrightarrow{t_{i,1}} P_1$ and $P_1 \xrightarrow{t_{i,2}} \epsilon$.*

Proof. In Algorithm 10, variables $s_1 = t'$ and $s_2 = t$. By Proposition 4.46, W_i is derived from P'.

Since $f_i = \texttt{true}$, by Algorithm 5, variable $t_i = \textsc{GetTrace}(\boxed{P'}, W_i, \sigma_i')$ for some σ_i'. By Theorem 2,

$$t_i \sim W_i, \tag{14}$$

$$P' \xrightarrow{t_i} \epsilon. \tag{15}$$

Since $P \xrightarrow{t} \epsilon$, by Figure 18, $Q \doteq Q_{k+1}$. Hence, by Figure 18,

$$P \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle} P_1. \tag{16}$$

Case 1: $t' = \text{Nil}$.

Variable $k = 0$ on line 6. Variable $t_{i,1} = \langle Q_1, r_{i,1}, \lambda_{i,1} \rangle$ on line 16. By Equation (16), $P \xrightarrow{t_{i,1}} P_1$.

Since $P' \xrightarrow{t'} P$, by Figure 18, $P' = P$. By Equation (15), $P \xrightarrow{t_i} \epsilon$.

Since $t_i = t_{i,1} \mathbin{@} t_{i,2}$, by Proposition 4.33, $P_1 \xrightarrow{t_{i,2}} \epsilon$.

Case 2: $t' \neq \text{Nil}$.

Variable $k \geq 1$ on line 6. Since $f_i = \texttt{true}$, variables $t_{i,1}$ and $t_{i,2}$ are defined on line 16 and satisfy:

$$t_i = t_{i,1} \mathbin{@} t_{i,2}. \tag{17}$$

Let $t_{i,1}' = \langle Q_1, r_{i,1}, \lambda_{i,1} \rangle, \ldots, \langle Q_k, r_{i,k}, \lambda_{i,k} \rangle$, then:

$$t_{i,1} = t_{i,1}' \mathbin{@} \langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle. \tag{18}$$

By Equation (14), Definition 3.19, and the definition of $\textsc{MakePathConstraint}$, $r_{i,j} = 0$ if and only if $r_j = 0$ for any $j = 1, \ldots, k$. Hence, traversing a program by following t' or by following $t_{i,1}'$ will use the same rules in Figure 18. Since $P' \xrightarrow{t'} P$,

$$P' \xrightarrow{t_{i,1}'} P. \tag{19}$$

By Equations (19), (16), (18), and Proposition 4.33,

$$P' \xrightarrow{t_{i,1}} P_1. \tag{20}$$

By Equations (15), (20), (17), and Proposition 4.33,

$$P_1 \xrightarrow{t_{i,2}} \epsilon. \tag{21}$$

□

Lemma 4.50. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$, where P is of the form "Seq", and any annotated traces t, t' such that $P' \xrightarrow{t'} P$ and $P \xrightarrow{t} \epsilon$. During the execution of* $\textsc{InferProg}(\boxed{P'}, t', t)$, *if variables $f_0 = f_1 = \texttt{true}$ on line 16, then either $t_{0,2} = t_{1,2} = Nil$, or $t_{0,2} \neq Nil$ and $t_{1,2} \neq Nil$ and $\pi_S Q_{0,k+2} = \pi_S Q_{1,k+2}$.*

Proof. Let P expand to "$Q\ P_1$", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol. By Lemma 4.49, $P_1 \xrightarrow{t_{i,2}} \epsilon$ for each $i = 0, 1$. The rest of the proof is by induction on the derivation of P_1.

Case 1: $P_1 = \epsilon$.

By Figure 18, $t_{i,2} = \text{Nil}$ for each $i = 0, 1$.

Case 2: P_1 is of the form “Seq”. P_1 expands to “Q' P_2”, where Q' corresponds to the Query symbol and P_2 corresponds to the Prog symbol.

By Figure 18, for each $i = 0, 1$, we have $t_{i,2} \neq \text{Nil}$ and $Q_{i,k+2} \doteq Q'$. Hence, $\pi_{\mathbb{S}} Q_{0,k+2} = \pi_{\mathbb{S}} Q_{1,k+2} = \pi_{\mathbb{S}} Q'$.

Case 3: P_1 is of the form “If”. The proof is similar to the proof of Case 2.

Case 4: P_1 is of the form “For”. The proof is similar to the proof of Case 2. □

Lemma 4.51. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$, where P is of the form “If”, and any annotated traces t, t' such that $P' \xrightarrow{t'} P$ and $P \xrightarrow{t} \epsilon$. Let P expand to “*`if` *Q*, `then` *P_1* `else` *P_2*”, where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. During the execution of* InferProg($\boxed{P'}$, t', t):

(1) if variable $f_0 = \texttt{true}$ on line 16, then $P' \xrightarrow{t_{0,1}} P_2$ and $P_2 \xrightarrow{t_{0,2}} \epsilon$.
(2) if variable $f_1 = \texttt{true}$ on line 16, then $P' \xrightarrow{t_{1,1}} P_1$ and $P_1 \xrightarrow{t_{1,2}} \epsilon$.

Proof.

(1) By Proposition 4.46, W_0 is derived from P'. Since $f_0 = \texttt{true}$, by Algorithm 5, variable $t_0 = \text{GetTrace}(\boxed{P'}, W_0, \sigma_0')$ for some σ_0'. By Theorem 2, $t_0 \sim W_0$. By the definition of MakePathConstraint, $r_{0,k+1} = 0$ on line 16. The rest of the proof is similar to the proof of Lemma 4.49.
(2) The proof is similar to the proof of 1. □

Lemma 4.52. *Consider any programs $P \in Prog$ and $P' \in \mathcal{K}$, where P is of the form “For”, and any annotated traces t, t' such that $P' \xrightarrow{t'} P$ and $P \xrightarrow{t} \epsilon$. Let P expand to “*`for` *Q* `do` *P_1* `else` *P_2*”, where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol. During the execution of* InferProg($\boxed{P'}$, t', t):

(1) if variable $f_0 = \texttt{true}$ on line 16, then $P' \xrightarrow{t_{0,1}} P_2$ and $P_2 \xrightarrow{t_{0,2}} \epsilon$.
(2) if variable $f_2 = \texttt{true}$ on line 16, then $P' \xrightarrow{t_{2,1}} P_1$ and $P_1 \xrightarrow{t_{2,2}} \epsilon$.

Proof. The proof is similar to the proof of Lemma 4.51. □

Theorem 3 (Core Recursion). *For any programs $P \in Prog$ and $P' \in \mathcal{K}$ and annotated traces t, t', if $P' \xrightarrow{t'} P$ and $P \xrightarrow{t} \epsilon$, then $P \doteq$* InferProg($\boxed{P'}$, t', t).

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

By Figure 18, $t = \text{Nil}$. By Algorithm 10, InferProg($\boxed{P'}$, t', Nil) $= \epsilon$.

Case 2: P is of the form “Seq”. P expands to “Q P_1”, where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

By Lemma 4.48, if $f_2 = \texttt{true}$, then $\lambda_{2,k+1} = \text{NotLoop}$, so execution does not enter the branch on line 17. By Lemma 4.50, execution does not enter the branch on line 23. Hence, execution enters the branch on line 27.

Since $P' \xrightarrow{t'} P$, by Figure 18, P is a subprogram of P'. Hence, Q is a query in P'. Since $P' \in \mathcal{K}$, by Definition 3.4 and Proposition 4.13, $\text{Trim}(P') = P'$. Hence, Q is a query in $\text{Trim}(P')$. By Proposition 4.24, there exists a context $\sigma \in Context$ such that Q is used while evaluating

$\sigma(P')$. So at least one of the path constraints W_0, W_1 is satisfiable. By Proposition 4.6, at least one of the variables f_0, f_1 is `true`.

If f_i = `true` ($i = 0, 1$), then by Lemma 4.49, $P' \xrightarrow{t_{i,1}} P_1$ and $P_1 \xrightarrow{t_{i,2}} \epsilon$. By the induction hypothesis, $P_1 \doteq \text{InferProg}(\boxed{P'}, t_{i,1}, t_{i,2})$. Either case, P_1 and variable b on line 31 are identical except for equivalent variables.

Since $P \xrightarrow{t} \epsilon$, by Figure 18, $Q \doteq Q_{k+1}$.

Case 3: P is of the form "If". P expands to "`if` Q, `then` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

By Lemma 4.48, if f_2 = `true`, then $\lambda_{2,k+1}$ = NotLoop, so execution does not enter the branch on line 17.

Since $P' \xrightarrow{t'} P$, by Figure 18, P is a subprogram of P'. Hence, Q is a query in P'. Since $P' \in \mathcal{K}$, by Definition 3.4 and Proposition 4.13, $\text{Trim}(P') = P'$. Hence, Q is a query in $\text{Trim}(P')$. By Proposition 4.24, there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(P')$ and the corresponding row count is zero (or positive). So both of the path constraints W_0, W_1 are satisfiable. By Proposition 4.6, variables $f_0 = f_1$ = `true`.

Since $\text{Trim}(P') = P'$ and P is subprogram of P', by Algorithm 8, it is not possible to have $P_1 = P_2 = \epsilon$. By Definition 3.4 and Definition 4.30, $\pi_{\mathcal{S}}\mathcal{F}(P_1) \neq \pi_{\mathcal{S}}\mathcal{F}(P_2)$.

By Lemma 4.51, $P' \xrightarrow{t_{0,1}} P_2$ and $P_2 \xrightarrow{t_{0,2}} \epsilon$. By Lemma 4.51, $P' \xrightarrow{t_{1,1}} P_1$ and $P_1 \xrightarrow{t_{1,2}} \epsilon$.

Case 3.1: $P_1 = \epsilon$ and $P_2 \neq \epsilon$.

By Figure 18, $t_{1,2}$ = Nil and $t_{0,2} \neq$ Nil.

Case 3.2: $P_1 \neq \epsilon$ and $P_2 = \epsilon$.

By Figure 18, $t_{1,2} \neq$ Nil and $t_{0,2}$ = Nil.

Case 3.3: $P_1 \neq \epsilon$ and $P_2 \neq \epsilon$.

By Figure 18, $t_{1,2} \neq$ Nil and $t_{0,2} \neq$ Nil. $Q_{0,k+2} \doteq \mathcal{F}(P_2)$. $Q_{1,k+2} \doteq \mathcal{F}(P_1)$. Since $\pi_{\mathcal{S}}\mathcal{F}(P_1) \neq \pi_{\mathcal{S}}\mathcal{F}(P_2)$, we have $\pi_{\mathcal{S}}Q_{0,k+2} \neq \pi_{\mathcal{S}}Q_{1,k+2}$.

In all of these cases, execution enters the branch on line 23.

By the induction hypothesis, P_2 and the variable $b_f = \text{InferProg}(\boxed{P'}, t_{0,1}, t_{0,2})$ on line 26 are identical except for equivalent variables. Also, P_1 and the variable $b_t = \text{InferProg}(\boxed{P'}, t_{1,1}, t_{1,2})$ are identical except for equivalent variables.

Since $P \xrightarrow{t} \epsilon$, by Figure 18, $Q \doteq Q_{k+1}$.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

Since $P' \xrightarrow{t'} P$, by Figure 18, P is a subprogram of P'. Hence, Q is a query in P'. Since $P' \in \mathcal{K}$, by Definition 3.4 and Proposition 4.13, $\text{Trim}(P') = P'$. Hence, Q is a query in $\text{Trim}(P')$. By Proposition 4.24, there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(P')$ and the corresponding row count is at least two. So the path constraint W_2 is satisfiable. By Proposition 4.6, variable f_2 = `true`.

By Lemma 4.48, variable $\lambda_{2,k+1} \neq$ NotLoop. Execution enters the branch on line 17.

By Lemma 4.52, $P' \xrightarrow{t_{2,1}} P_1$ and $P_1 \xrightarrow{t_{2,2}} \epsilon$. By the induction hypothesis, P_1 and the variable $b_t = \text{InferProg}(\boxed{P'}, t_{2,1}, t_{2,2})$ on line 22 are identical except for equivalent variables.

Case 4.1: f_0 = `true`.

By Lemma 4.52, $P' \xrightarrow{t_{0,1}} P_2$ and $P_2 \xrightarrow{t_{0,2}} \epsilon$. By the induction hypothesis, P_2 and the variable $b_f = \textsc{InferProg}(\boxed{P'}, t_{0,1}, t_{0,2})$ on line 22 are identical except for equivalent variables.

Case 4.2: $f_0 = \texttt{false}$.

The path constraint W_0 is unsatisfiable. Since $\textsc{Trim}(P') = P'$, by Algorithm 8, $P_2 = \epsilon$. Hence, $P_2 = b_f$ on line 22.

Since $P \xrightarrow{t} \epsilon$, by Figure 18, $Q \doteq Q_{k+1}$. □

4.5.3 *Soundness of* Infer.

Theorem 4 (Soundness of Inference). *For any program $P \in \mathcal{K}$, $P \doteq \textsc{Infer}(\boxed{P})$.*

Proof. By Definition 3.16, the variable σ in Algorithm 1 satisfies the trivial path constraint Nil. By Figure 18, there exists an annotated trace t' such that $P \xrightarrow{t'} \epsilon$. By Definition 3.19, $t' \sim \text{Nil}$. By Definition 4.4, the trivial path constraint Nil is derived from P.

Since $P \in \mathcal{K}$, by Theorem 2, variable t satisfies $P \xrightarrow{t} \epsilon$. By Figure 18, $P \xrightarrow{\text{Nil}} P$. By Theorem 3, $P \doteq \textsc{InferProg}(\boxed{P}, \text{Nil}, t)$. □

Corollary 4.53. *For any programs $P_1, P_2 \in \mathcal{K}$, if $P_1 \equiv P_2$, then $P_1 \doteq P_2$.*

Proof. By Proposition 4.15, $\textsc{Infer}(\boxed{P_1}) = \textsc{Infer}(\boxed{P_2})$. Since $P_1, P_2 \in \mathcal{K}$, by Theorem 4, $P_1 \doteq \textsc{Infer}(\boxed{P_1})$ and $P_2 \doteq \textsc{Infer}(\boxed{P_2})$. □

Corollary 4.54. *For any programs $P_1, P_2 \in \mathcal{K}$, $P_1 \equiv P_2$ if and only if $P_1 \doteq P_2$.*

Proof. By Proposition 4.20 and Corollary 4.53. □

4.6 Complexity

We show that the number of recursive calls to Algorithm 6 is linear in the size of the given program.

Lemma 4.55. *For any programs $P \in Prog$ and $P' \in \mathcal{K}$ and annotated traces t, t', if $P' \xrightarrow{t'} P$ and $P \xrightarrow{t} \epsilon$, then the execution of $\textsc{InferProg}(\boxed{P'}, t', t)$ calls the* InferProg *procedure at most $(\|P\| - 1)$ times.*

Proof. This proof is by induction on the derivation of P.

Case 1: $P = \epsilon$.

By Figure 18, $t = \text{Nil}$. By Algorithm 6, the procedure returns immediately without calling InferProg. By Definition 4.5, $\|\epsilon\| = 1$.

Case 2: P is of the form "Seq". P expands to "Q P_1", where Q corresponds to the Query symbol and P_1 corresponds to the Prog symbol.

By the proof of Theorem 3, execution in Algorithm 6 enters the branch on line 24. This branch calls the InferProg procedure once. At least one of the variables f_0, f_1 is $\texttt{true}$. When $f_i = \texttt{true}$ $(i = 0, 1)$, by the induction hypothesis, $\textsc{InferProg}(\boxed{P'}, t_{i,1}, t_{i,2})$ recursively calls the InferProg procedure at most $(\|P_1\| - 1)$ times. Either case, InferProg is totally called at most $1 + (\|P_1\| - 1) = \|P_1\|$ times. By Definition 4.5, $\|P\| = 1 + \|P_1\|$.

Case 3: P is of the form "If". P expands to "$\texttt{if}$ Q, $\texttt{then}$ P_1 $\texttt{else}$ P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

By the proof of Theorem 3, execution in Algorithm 6 enters the branch on line 20. This branch calls the InferProg procedure twice. By the induction hypothesis, InferProg($\boxed{P'}$, $t_{0,1}, t_{0,2}$) calls the InferProg procedure at most ($\|P_2\| - 1$) times and InferProg($\boxed{P'}$, $t_{1,1}, t_{1,2}$) calls the InferProg procedure at most ($\|P_1\| - 1$) times. Hence, InferProg is totally called at most $2 + (\|P_1\| - 1) + (\|P_2\| - 1) = \|P_1\| + \|P_2\|$ times. By Definition 4.5, $\|P\| = 1 + \|P_1\| + \|P_2\|$.

Case 4: P is of the form "For". P expands to "`for` Q `do` P_1 `else` P_2", where Q corresponds to the Query symbol, P_1 corresponds to the first Prog symbol, and P_2 corresponds to the second Prog symbol.

By the proof of Theorem 3, execution in Algorithm 6 enters the branch on line 14. This branch calls the InferProg procedure at most twice. The rest of the proof is similar to the proof of Case 3. □

Theorem 5 (Complexity). *For any program $P \in \mathcal{K}$, the execution of Infer($\boxed{P}$) calls the InferProg procedure at most $\|P\|$ times.*

Proof. By the proof of Theorem 4, we have $P \xrightarrow{\text{Nil}} P$ and $P \xrightarrow{t} \epsilon$ for the variable t in Algorithm 1. By Lemma 4.55, the execution of InferProg($\boxed{P}$, Nil, t) recursively calls the InferProg procedure at most ($\|P\| - 1$) times. By Algorithm 1, the execution of Infer($\boxed{P}$) directly calls the InferProg procedure once. Hence, InferProg is totally called at most $1 + (\|P\| - 1) = \|P\|$ times. □

5 REMARK ON THE Konure DSL

We next discuss the outcomes of using Konure to infer programs that are not in $\mathcal{K}$.

5.1 Programs in Konure DSL Grammar

Apart from the set of inferrable programs $\mathcal{K}$ (Definition 3.4) for which we designed Konure, we also identify the following interesting sets of programs in Prog, where we obtain a stronger result.

Definition 5.1.

$$K_2 = \{P \mid P \in \text{Prog}, \widetilde{P} \in \mathcal{K}\},$$
$$K_3 = \{P \mid P \in \text{Prog}, \exists P' \in \mathcal{K} : P \equiv P'\},$$
$$K_4 = \{P \mid P \in \text{Prog}, \text{Infer}(\boxed{P}) \equiv P\}.$$

K_2 represents the set of programs in Prog for which the Trim transformation produces an equivalent program in $\mathcal{K}$. K_3 represents the set of programs in Prog that have an equivalent program in $\mathcal{K}$ but the Trim transformation may not necessarily produce the program in $\mathcal{K}$. K_4 represents the set of programs in Prog that Infer is able to infer correctly, although it is not designed to support these programs (because our Konure DSL restrictions are conservative).

Corollary 5.2. *For any programs $P_1, P_2 \in K_2$, if $P_1 \equiv P_2$, then $\widetilde{P_1} \doteq \widetilde{P_2}$.*

Proof. By Definition 3.2 and Theorem 7, $\widetilde{P_1} \equiv P_1$ and $\widetilde{P_2} \equiv P_2$. By Definition 4.14, $\widetilde{P_1} \equiv \widetilde{P_2}$. By the definition of K_2, $\widetilde{P_1}, \widetilde{P_2} \in \mathcal{K}$. By Corollary 4.53, $\widetilde{P_1} \doteq \widetilde{P_2}$. □

Corollary 5.3. *For any program $P \in K_3$, let $P' \in \mathcal{K}$ such that $P \equiv P'$, then $P' \doteq$ Infer($\boxed{P}$).*

Proof. By the definition of K_3, such program P' exists. Since $P \equiv P'$, by Proposition 4.15, Infer($\boxed{P}$) = Infer($\boxed{P}'$). By Theorem 4, $P' \doteq$ Infer($\boxed{P'}$). □

We distinguish the sets $\mathcal{K}$, K_2, K_3, K_4, and Prog as follows:

Proposition 5.4. $\mathcal{K} \subset K_2$.

Proof.

(1) $\mathcal{K} \subseteq K_2$: For any program $P \in \mathcal{K}$, by Definition 3.4, there exists program $P' \in$ Prog such that $P = \widetilde{P'}$. By Definition 3.2 and Proposition 4.13, $\widetilde{\widetilde{P'}} = \widetilde{P'}$. In other words, $\widetilde{P} = P \in \mathcal{K}$. Hence, $P \in K_2$.

(2) $\mathcal{K} \neq K_2$: Consider the following example: Let queries $Q_1, Q_2 \in$ Query such that $\pi_S Q_1 \neq \pi_S Q_2$ and that there exists contexts $\sigma, \sigma' \in \mathit{Context}$ such that Q_1 retrieves nonempty data with σ and retrieves empty data with σ'. Let program $P \in$ Prog be as follows:

$$P = \texttt{if } Q_1, \texttt{ then } \{\texttt{if } Q_1, \texttt{ then } Q_2 \texttt{ else } \epsilon\} \texttt{ else } \epsilon.$$

By Definition 3.2,

$$\widetilde{P} = \texttt{if } Q_1, \texttt{ then } \{Q_1\ Q_2\} \texttt{ else } \epsilon.$$

By Definition 3.4, $\widetilde{P} \in \mathcal{K}$. By the definition of K_2, $P \in K_2$. Since $P \neq \widetilde{P}$, by Proposition 4.13, there does not exist any program $P' \in$ Prog such that $P = \widetilde{P'}$. By Definition 3.4, $P \notin \mathcal{K}$. □

Proposition 5.5. $K_2 \subset K_3$.

Proof.

(1) $K_2 \subseteq K_3$: For any program $P \in K_2$, by definition, $\widetilde{P} \in \mathcal{K}$. By Definition 3.2 and Theorem 7, $P \equiv \widetilde{P}$. Hence, $P \in K_3$.

(2) $K_2 \neq K_3$: Consider the following example: Let queries $Q_1, Q_2, Q_3, Q_4 \in$ Query such that $\pi_S Q_1, \pi_S Q_2, \pi_S Q_3, \pi_S Q_4$ are distinct and that there exists contexts $\sigma, \sigma' \in \mathit{Context}$ such that Q_1 retrieves nonempty data with σ and retrieves empty data with σ'. Let programs $P_1, P_2 \in$ Prog be as follows:

$$P_1 = Q_1\ Q_2\ \texttt{if } Q_1, \texttt{ then } Q_3 \texttt{ else } Q_4,$$
$$P_2 = \texttt{if } Q_1, \texttt{ then } \{Q_2\ Q_1\ Q_3\} \texttt{ else } \{Q_2\ Q_1\ Q_4\}.$$

By Definition 4.14, $P_1 \equiv P_2$. By Definition 3.2, $\widetilde{P_1} = P_1$ and $\widetilde{P_2} = P_2$. By Definition 3.4, $P_1 \in \mathcal{K}$ and $P_2 \notin \mathcal{K}$. Hence, $P_2 \in K_3$ and $P_2 \notin K_2$. □

Proposition 5.6. $K_3 \subset K_4$.

Proof.

(1) $K_3 \subseteq K_4$: For any program $P \in K_3$, by definition, there exists program $P' \in \mathcal{K}$ such that $P \equiv P'$. By Theorem 4, $P' \doteq \textsc{Infer}(\boxed{P'})$. By Proposition 4.20, $P' \equiv \textsc{Infer}(\boxed{P'})$. By Definition 4.14, $\textsc{Infer}(\boxed{P'}) \equiv P$.

(2) $K_3 \neq K_4$: Consider the following program $P \in$ Prog.

```
if y1 ← select * from t1 where t1.val1 = x1 {
    y2 ← select * from t2 where t2.val1 = y1.t1.val1
    y3 ← select * from t2 where t2.id = x1
    if y4 ← select * from t1 where t1.val1 = x1 ∧ t1.val2 = x2 {
        if y5 ← select * from t1 where t1.val1 = x1 ∧ t1.val2 = x3 {
            for y6 ← select * from t1 where t1.val1 = x1 {
                y7 ← select * from t2 where t2.val1 = y6.t1.val1
            } else {}
        } else {}
    } else {}
} else {}
```

Variables x_1, x_2, x_3 are distinct input parameters. Table `t1` has columns `val1` and `val2`. Table `t2` has columns `id` and `val1`, where `id` is the primary key.

By Definition 3.2, $\widetilde{P} = P$. Since query y_1 may return more than one row, $y_1 \in \mathcal{T}(\widetilde{P})$. Since queries y_2 and y_7 have the same skeleton, $y_1 \in \mathcal{R}(\widetilde{P})$. Hence, $\mathcal{T}(\widetilde{P}) \cap \mathcal{R}(\widetilde{P}) \neq \emptyset$. By Definition 3.4, $\widetilde{P} \notin \mathcal{K}$. Hence, $P \notin \mathcal{K}$ and $P \notin K_2$.

To show that $P \in K_4$, we first show that the loop detection algorithm in Algorithm 3 correctly identifies loops for P. Let $r_i = |y_i|$ be the number of rows retrieved by query y_i for each $i = 1, 2, \ldots, 7$. When execution enters query y_6, we have $r_1 > 0$, $r_4 > 0$, and $r_5 > 0$. Hence, $y_1 \neq \emptyset$, $y_4 \neq \emptyset$, and $y_5 \neq \emptyset$. Note that the rows retrieved by y_4 and y_5 are both subsets of the rows retrieved by y_1, that is, $y_4 \subseteq y_1$ and $y_5 \subseteq y_1$. Since x_2 and x_3 are distinct input parameters, the Konure inference algorithm assigns them different values (Section 3.4). Hence, the rows retrieved by y_4 and y_5 are disjoint, that is, $y_4 \cap y_5 = \emptyset$. Since y_4 and y_5 are both nonempty, we have $y_4 \subset y_1$ and $y_5 \subset y_1$. Hence, $r_1 > r_4 > 0$, $r_1 > r_5 > 0$, and $r_1 \geq 2$. Since queries y_1 and y_6 are identical, $r_1 = r_6 \geq 2$. Since query y_7 is repeated $r_6 \geq 2$ times in the trace, the loop detection algorithm in Algorithm 3 correctly identifies query y_7 as iterations of a loop that iterates over query y_6.

We next discuss the two other sets of repetitive query skeletons:

(a) Queries y_2 and y_7 have the same skeleton. During execution, this skeleton is repeated $(r_6 + 1)$ times in the trace. Since $r_6 + 1 = r_1 + 1 \neq r_1$, the loop detection algorithm does not incorrectly identify queries y_1 and y_7 as iterations of a loop that iterates over query y_1.

(b) Queries y_4 and y_5 have the same skeleton. Since query y_3 selects data by the primary key, $r_3 \leq 1$. Hence, the loop detection algorithm does not incorrectly identify queries y_4 and y_5 as iterations of a loop that iterates over query y_3.

For these reasons, the DetectLoops procedure is able to infer the correct loop layout trees. The rest of the Konure inference algorithm produces Infer($\boxed{P}$), where $P \doteq$ Infer($\boxed{P}$). By Proposition 4.20, $P \equiv$ Infer($\boxed{P}$). By the definition of K_4, $P \in K_4$.

To show that $P \notin K_3$, assume by way of contradiction that $P \in K_3$. By the definition of K_3, there exists $P' \in \mathcal{K}$ such that $P \equiv P'$. By Theorem 4, $P' \doteq$ Infer($\boxed{P'}$). Since $P \equiv P'$, the black box programs $\boxed{P}$ and $\boxed{P'}$ are observationally equivalent. Hence, Infer($\boxed{P}$) = Infer($\boxed{P'}$). Since $P \doteq$ Infer($\boxed{P}$), we have $P \doteq P'$. No matter how we alter P with different but equivalent origin locations, the query y_1 may still return more than one row and the queries y_2 and y_7 still have the same skeleton. Hence, $\mathcal{T}(P') \cap \mathcal{R}(P') \neq \emptyset$. Since $P' \in \mathcal{K}$, we have the desired contradiction. Hence, $P \notin K_3$. □

Proposition 5.7. *$K_4 \subset$ Prog.*

Proof.

(1) $K_4 \subseteq$ Prog: By definition.
(2) $K_4 \neq$ Prog: Consider the following example: Let queries $Q_1, Q_2 \in$ Query such that $\pi_{\mathbb{S}} Q_1$ and $\pi_{\mathbb{S}} Q_2$ are distinct and that there exists context $\sigma \in$ *Context* such that Q_1 retrieves two rows with σ. Let program $P \in$ Prog be as follows:

$$P = Q_1\ Q_2\ Q_2.$$

The execution of Infer($\boxed{P}$) may fail because the loop detection algorithm in Algorithm 3 may observe Q_1 retrieve two rows in an execution and mistakenly identify the two subsequent Q_2 queries as two iterations of a loop. Hence, $P \notin K_4$. □

Proposition 5.4 states that the Trim transformation transforms certain programs that are not in the Konure DSL into equivalent programs in the Konure DSL. Proposition 5.5 states that the Trim transformation does not transform all of the potential programs into the Konure DSL. Proposition 5.6 states that the restrictions in Definition 3.4 are conservative, that is, there are programs not expressible in the Konure DSL but still allows the Konure inference algorithm to infer the correct program. Proposition 5.7 states that the Konure DSL syntax alone is not sufficient for inferrability.

5.2 Programs Expressible in Konure DSL

Recall that two programs in Prog are observationally equivalent (Definition 4.14) if they produce the same concrete trace (Definition 3.8) for all contexts. In other words, when these programs are executed as black boxes (Definition 3.9), they always produce the same list of SQL queries and the same retrieved rows. These concrete traces are the only behavior directly observed by Konure in the Execute procedure. We extend our results to black box programs that are not necessarily written in the Konure DSL grammar but share the externally visible behavior of some program in $\mathcal{K}$.

Definition 5.8. Ⓤ denotes the black box executable for a program with an unknown implementation. To execute Ⓤ with a context $\sigma \in$ *Context*, we populate the database, set the input parameters, and collect the concrete trace as in the Execute procedure.

Definition 5.9. Ⓤ is *expressible* as program $P \in$ Prog if for all contexts $\sigma \in$ *Context*, executing Ⓤ with σ produces $\sigma(P)$. Ⓤ is expressible in $\mathcal{K}$ if there exists a program $P \in \mathcal{K}$ such that Ⓤ is expressible as P.

Proposition 5.10. *For any program $P \in K_3$, $\boxed{P}$ is expressible in $\mathcal{K}$.*

Proof. By the definition of K_3 (Definition 5.1), there exists program $P' \in \mathcal{K}$ such that $P \equiv P'$. By Definition 4.14, for any context $\sigma \in$ *Context*, $\sigma(P) = \sigma(P')$. By Definition 3.9, executing $\boxed{P}$ produces $\sigma(P')$. By Definition 5.9, $\boxed{P}$ is expressible as P' and is expressible in $\mathcal{K}$. □

Proposition 5.11. *For any program $P \in Prog$, if Ⓤ is expressible as P, then* Infer(Ⓤ) = Infer($\boxed{P}$).

Proof. By Definition 5.9, for any context $\sigma \in$ *Context*, Execute(Ⓤ, σ) = Execute($\boxed{P}$, σ). By Algorithm 1, Infer(Ⓤ) = Infer($\boxed{P}$). □

Corollary 5.12. *For any program $P \in \mathcal{K}$, if Ⓤ is expressible as P, then $P \doteq$ Infer(Ⓤ).*

Proof. By Proposition 5.11 and Theorem 4. □

Corollary 5.12 states that, as long as the program executable is expressible in $\mathcal{K}$, Konure infers it correctly. The program can be implemented in arbitrary languages or programming styles.

Example programs that can be expressible in Konure DSL include the data retrieval components of task managers, blogs, chat rooms, and inventory management systems. In practice, most of the real-world programs, even if expressible in the Konure DSL, are implemented in standard programming languages such as Java, Ruby, and Python. Because of our black box approach, Konure can work with these programs as long as their externally visible behavior conforms to the Konure DSL.

6 EXPERIMENTAL RESULTS

We implemented a Konure prototype and acquired five benchmark applications to evaluate this prototype. Each application has multiple commands that access different parts of the database.

Each command takes input parameters, translates the inputs into SQL queries against the relational database, and returns results extracted from the results of the queries.

6.1 Applications and Commands

Our benchmark applications include:

- **Fulcrum Task Manager:** Fulcrum [2] is an open source project planning tool, built with Ruby on Rails, with over 1,500 stars on GitHub. Fulcrum maintains multiple projects. Each project may contain multiple stories. Each story may contain multiple notes. Fulcrum commands enable users to navigate the contents of projects, stories, and notes, as well as the users who created these contents.
- **Kandan Chat Room:** Kandan [4] is an open source chat room application, built with Ruby on Rails, with over 2,700 stars on GitHub. Kandan maintains multiple chat rooms (so-called channels) that users can access. Its commands enable users to navigate chat rooms and messages (so-called activities) and display relevant user information.
- **Enki Blogging Application:** Enki [1] is an open source blogging application, built with Ruby on Rails, with over 800 stars on GitHub. Enki maintains multiple pages and posts, each of which may have comments. Enki commands enable the author of the blog to navigate pages, posts, and comments.
- **Blog:** The Blog application is an example obtained from the Ruby on Rails website [3]. Blog maintains information about blog articles and blog comments. It implements a command that retrieves all articles and a command that retrieves a specific article and its associated comments.
- **Student Registration:** The student registration application discussed in Section 2. This application was adapted from an earlier version of a program developed by the MITRE Corporation. The version was developed specifically for studying the detection and nullification of SQL injection attacks. In the test suite titled "IARPA STONESOUP Phase 1 - Injection for Java" [6], the version "TC_Java_89_m100" is the most similar to the program that we used and implements largely the same functionality.

The Fulcrum, Enki, and Blog servers receive HTTP requests, interact with the database accordingly, and respond the client with an HTML page that contains the data retrieved. The Kandan server receives HTTP requests, interacts with the database accordingly, and responds with JSON objects that contain data retrieved and HTML templates to display the JSON data. For these applications, the Konure prototype works with the retrieved database results after they are automatically extracted from the surrounding HTML/JSON code. Student Registration implements a command-line interface that receives text commands, interacts with the database accordingly, and responds with text output.

Application Selection Criteria: We choose our real-world benchmark applications—Fulcrum, Kandan, and Enki—from the applications studied in a recent survey paper [90]. We choose these three applications because their core functionality shares a common pattern, as characterized by the Konure DSL. We omit other applications in the survey mainly for three reasons: (1) In some applications, the control flow and the data flow are similar to that of the Konure DSL. However, these applications perform computations that are more complicated than the Konure DSL currently supports. Such computations often belong to standard domains such as string manipulation, aggregate calculation, and date/time conversion. Example applications include task managers, chat rooms, and blogs with more complicated features than Fulcrum, Kandan, and Enki. To support these applications, we anticipate that the solver for Konure would need to incorporate more knowledge to work productively with a number of standard domains. (2) Some applications implement highly specialized calculations. For example, online shopping applications perform specific numeric calculations specific to that domain. (3) In some applications, the control flow does not depend primarily on the results of database queries. Example applications include file sharing applications whose control logic relies heavily on the state of the file system. To support these applications, we anticipate that Konure would need to observe the file system traffic and incorporate the file system operations into the active learning algorithm. The remaining benchmark applications—Blog and Student—implement interesting core functionality that is expressible in the Konure DSL.

Based on our understanding and use of the applications, we identified data retrieval commands that these applications execute as part of their standard functionality. In general, these commands step through tables, typically using results from earlier look-ups to access the correct data in current tables. As a command traverses tables, it collects data to return to the user. Fulcrum uses five database tables, Kandan uses four database tables, Enki uses five database tables, Blog uses two database tables, and Student Registration uses five database tables. For Fulcrum, we identified 8 of 14 data retrieval commands as potential inference candidates. For Kandan, we identified 6 of 11, for Enki, 4 of 10, for Blog, 2 of 2, and for Student Registration, 1 of 1. The remaining commands in these applications often implement specialized data or control flow that are not expressible in the Konure DSL. We discuss unsupported commands in Section 6.2.

Results: We built virtual machines for executing these applications, then configured our Konure prototype to operate properly in this context. Specifically, the Rails framework stores password hashes in the database. Based on the Rails configuration, the Rails framework uses these hashes to perform a password check at the start of specified commands. We configured our Konure prototype to generate databases and parameters that, during inference, always pass the password check. We also support the insertion of boilerplate password checking code into the regenerated code for specified commands. We anticipate that the automated introduction of such boilerplate code will be standard in many usage contexts. We then used Konure to infer and regenerate the commands. The source code for the regenerated commands is available in the Appendix and Reference [5].

Table 1 presents statistics from running the Konure prototype on the commands. The first column (**Command**) presents the name of the command. The second (**Params**) presents the number of input parameters for the command. The third (**App**) presents the name of the application.

The next column (**Runs**) presents the number of executions that Konure used to infer the command. Each execution involves a set of generated input values presented to the application working with generated database contents. All commands require fewer than 30 executions to obtain a model for the command as expressed in the Konure DSL. The next column (**Solves**) presents the number of invocations of the Z3 SMT solver that Konure executed to infer the model for the command. Because Konure may invoke the SMT solver multiple times for each inference step, the number of Z3 invocations is larger than the number of application executions. The next column (**Time**) presents the wall-clock time required to infer the model for each command. We measured time on a Ubuntu 16.04 virtual machine with 2 cores and 2 GB memory. The host machine uses a processor with 4 cores (3.4 GHz Intel Core i5) and has 24 GB 1600 MHz DDR3 memory. The times vary from less than a minute to about two hours. In general, the times are positively correlated with the number of solves, the length of the programs, and the number of potentially ambiguous origin locations. Most of the inference time was spent on solving for alternative database contents to satisfy various constraints. The inference time also includes the time required to set up, tear down, and execute the applications (and their web servers) in the Konure environment.

The remaining columns present statistics from the regenerated Python implementations. The **Regen** column presents the Appendix that contains the regenerated Python implementation. The

Table 1. Inference Effort and Regenerated Code Size

Command	**Params**	**App**	**Runs**	**Solves**	**Time**	**Regen**	**LoC**	**SQL**	**If**	**For**	**Output**
`get_home`	1	Fulcrum	5	43	8 mins	Appendix B.1	21	5	1	0	9
`get_projects`	1	Fulcrum	5	43	8 mins	Appendix B.2	21	5	1	0	9
`get_projects_id`	2	Fulcrum	12	124	29 mins	Appendix B.3	25	8	2	0	8
`get_projects_id_stories`	2	Fulcrum	11	42	7 mins	Appendix B.4	31	8	3	0	11
`get_projects_id_stories_id`	3	Fulcrum	12	50	8 mins	Appendix B.5	31	9	3	0	11
`get_projects_id_stories_id_notes`	3	Fulcrum	11	41	8 mins	Appendix B.6	24	9	3	0	4
`get_projects_id_stories_id_notes_id`	4	Fulcrum	13	46	10 mins	Appendix B.7	28	10	4	0	4
`get_projects_id_users`	2	Fulcrum	12	124	30 mins	Appendix B.8	25	8	2	0	8
`get_channels`	1	Kandan	21	125	105 mins	Appendix C.1	63	16	4	2	27
`get_channels_id_activities`	2	Kandan	23	242	39 mins	Appendix C.2	49	16	6	0	13
`get_channels_id_activities_id`	3	Kandan	14	18	7 mins	Appendix C.3	25	11	3	0	3
`get_me`	1	Kandan	11	139	6 mins	Appendix C.4	44	8	3	0	25
`get_users`	1	Kandan	15	236	9 mins	Appendix C.5	67	11	3	0	45
`get_users_id`	2	Kandan	11	139	6 mins	Appendix C.6	44	8	3	0	25
`get_admin_comments_id`	1	Enki	2	5	22 secs	Appendix D.1	10	1	0	0	5
`get_admin_pages`	0	Enki	2	1	22 secs	Appendix D.2	13	2	1	0	4
`get_admin_pages_id`	1	Enki	2	5	23 secs	Appendix D.3	9	1	0	0	4
`get_admin_posts`	0	Enki	3	2	33 secs	Appendix D.4	16	3	1	1	3
`get_articles`	0	Blog	2	11	21 secs	Appendix E.1	12	2	0	0	6
`get_article_id`	1	Blog	6	29	42 secs	Appendix E.2	16	3	1	0	6
`liststudentcourses`	2	Student	6	20	41 secs	Appendix F.1	24	5	3	1	3

LoC, **SQL**, **If**, **For**, and **Output** columns present the number of lines of code, SQL statements, If statements, For statements, and the number of lines that generate output.

Quality of the Regenerated Code: We recruited a software engineer with three years of experience working with Ruby on Rails applications to evaluate the Konure inference and regeneration by comparing the original Ruby on Rails and regenerated Python versions of each command. Starting from a command URL, the software engineer locates the relevant controller, models, and views in the original Ruby on Rails application to form an understanding of the program functionality. The software engineer mentally translates the Ruby on Rails abstractions into concrete actions and compares them against the regenerated Python code. (1) One complication was that the Ruby on Rails framework automatically generates a substantial amount of database traffic that is not directly reflected in the Ruby on Rails code. This traffic was explicitly reflected in the regenerated code. The software engineer was occasionally surprised to see these queries in the regenerated code, but eventually understood that they accurately reflect the low-level implementation of the high-level aspect abstractions in Ruby on Rails. (2) Another complication was that the Ruby on Rails implementation contains auxiliary functionality (such as session management) that performs database queries and checks the query results against specific values (such as checking if the user is an admin). Our Konure implementation captures these database queries and includes them in the regenerated code, but does not currently regenerate the associated conditional checks against the specific values. After taking these phenomena into account, the software engineer determined that the regenerated commands were consistent with the original Ruby on Rails implementations.

The evaluation also highlights how the Rails framework, specifically the ActiveRecord object relational mapping abstraction, implicitly generates substantial database traffic as it assembles the object state (including the state of objects on which it depends) when initially loading the object. This code that generates this database traffic is explicit and therefore directly visible in the regenerated Python code.

This comparison of the original Ruby on Rails code with the regenerated version highlights two key properties of the regenerated version. (1) Understandability: Because the regenerated Python code performs database queries explicitly, we anticipate that the regenerated code can help developers comprehend the program behavior at the level of database queries. (2) Streamlined implementation: The regenerated code contains only the core functionality as expressed in the Konure DSL and does not need to implement the less common features that are required in comprehensive abstraction frameworks such as Ruby on Rails. As a result, the regenerated program is often lighter weight than the original application.

Noisy Specifications: We note that the regenerated programs are free of SQL injection attack vulnerabilities, as Konure regenerates programs using a standard SQL library in Python that systematically eliminates the possibility of these attacks. However, these vulnerabilities are present in the original student registration application. These vulnerabilities are rare corner cases that are not captured by the Konure DSL. Thus, Konure omits them and infers only the common use cases of the program. These results highlight the ability of Konure to work with noisy specifications.

6.2 Commands Not Expressible in Konure DSL

In our experiments, we observed data-retrieval commands that are not fully expressible in the Konure DSL. For example, several Enki commands condition on whether a retrieved value is "NULL" (undetected conditionals). Several other Enki commands combine multiple input parameters before using the combined value to access the database (unanticipated data calculations). A Kandan command produces inconsistent traces even if the path constraints in the Konure inference algorithm remain unchanged (unanticipated control flow). In addition to these real-world applications and commands, we also developed an adversarial synthetic program that may cause non-termination of the Konure inference algorithm. We used Konure to infer these commands and report the outcomes below with representative examples.

Undetected Conditionals Outside Konure DSL (Omitted Functionality): Recall from Sections 3.1.3 and 3.1.4 that Konure is designed to infer control structures that depend largely on externally observable data, specifically, the database queries and results. A program that is not expressible in the Konure DSL may contain a conditional statement that, after retrieving data from the database, compares a retrieved value against a specific constant value (such as "NULL", "1", or "admin"). Konure is not designed to generate the specific inputs and database values for inferring conditional statements of this form, especially when the conditional checks are not externally observable. As a result Konure may infer a slice of the program functionality that conforms to the Konure DSL, omitting the undetected branches, without reporting any errors.

Omitting functionality in this form enables Konure to work with noisy specifications. Konure is likely to work well with programs whose main functionality is expressible in the Konure DSL with exceptions on rare corner cases. For example, if a program is defective when handling rare corner case inputs (and database values), Konure is likely to omit the functionality for the rare corner cases and end up inferring only the main functionality.

Example 19. Consider the following Python program inspired by the applications in our experiments:

```
def outside(conn, inputs):
  s1 = util.do_sql(conn, 'SELECT * FROM t1 WHERE id = :x', {'x': inputs[0]})
  if util.has_rows(s1):
    v = util.get_one_data(s1, 't1', 'val')
    print(v)
    if v == 0:
      crash()
    else:
      s2 = util.do_sql(conn, 'SELECT * FROM t2 WHERE id = :x', {'x': v})
```

The database has two tables, `t1` and `t2`. Each table has two columns, `id` and `val`, both holding integers. The column `id` of each table is the unique primary key. The `conn` variable is an established database connection. The `inputs` variable holds the list of input parameters. This program uses one input parameter, `inputs[0]`. The call to `util.do_sql` first assembles an SQL query by replacing "`:x`" with the value of the input parameter, then performs this query on the database, and finally stores the retrieved rows in variable `s1`. The call to `util.has_rows` checks whether variable `s1` holds nonempty rows. When `s1` holds nonempty rows, the call to `util.get_one_data` extracts from this row the integer in column `val` and stores it in variable `v`. After printing the value of `v`, the program behaves differently, depending on whether this value equals a constant number, zero. Depending on this check, the program either crashes or proceeds to perform another query. This conditional check is not directly observable in the database traffic and causes the program to be not expressible in Konure DSL.

When inferring this program, our current Konure implementation does not generate database values that cause variable `v` to equal zero. As a result this program never enters the corresponding branch. Konure thus infers and regenerates a slice of this program that performs the second query regardless of the value of `v`. During this inference, Konure does not report any errors.

Error Reported for Unanticipated Control Flow Behavior: We designed Konure to work with programs expressible in the Konure DSL. For example, the inference algorithm assumes that all conditional statements in the program must condition on query results being empty or nonempty. In other words, if a query produces the same empty/nonempty results across two executions of the program, the program should continue to execute the same path in both executions. This assumption does not hold for programs that are not expressible in the Konure DSL. For these programs, different executions may behave inconsistently, depending on unanticipated factors. In this case, Konure may detect the unanticipated behavior, report an error, and exit prematurely.

Example 20. Consider the following Python program inspired by the applications in our experiments:

```
def outside(conn, inputs):
  s1 = util.do_sql(conn, 'SELECT * FROM t1 WHERE id = :x', {'x': inputs[0]})
  if rand():
    s2 = util.do_sql(conn, 'SELECT * FROM t2 WHERE id = :x', {'x': inputs[1]})
    print(s2)
  if rand():
    s3 = util.do_sql(conn, 'SELECT * FROM t2 WHERE val = :x', {'x': inputs[2]})
    print(s3)
  if rand():
    s4 = util.do_sql(conn, 'SELECT * FROM t1 WHERE id = :x', {'x': inputs[1]})
    print(s4)
```

The database has two tables, `t1` and `t2`. Each table has two columns, `id` and `val`, both holding integers. The columns `id` are the unique primary keys. The `conn` variable is an established database connection. The `inputs` variable holds the list of input parameters. This program uses three input parameters, `inputs[0]`, `inputs[1]`, and `inputs[2]`. Each call to `util.do_sql` first assembles an SQL query by replacing "`:x`" with the value of the specified input parameter, then performs this query on the database. The retrieved rows are then stored in the corresponding variable, `s1`, `s2`, `s3`, or `s4`. Each call to `rand` obtains a random Boolean value, either True or False. Conditioned on these random values, the program may or may not execute the branches that perform queries for `s2`, `s3`, and `s4`. We use the `rand` function to emulate the effects of uninferrable conditional expressions that are not captured by the Konure DSL.

When inferring this program, our current Konure implementation often observes two inconsistent executions. Both executions perform the query for `s2` and retrieve empty data. However, in one execution the next query is the query for `s3`, while in the other execution the next query is the query for `s4`. This behavior is not expressible in the Konure DSL, which triggers an assertion failure in our current Konure implementation.

Error Reported for Unanticipated Data Calculations: We designed the Konure DSL to express programs whose data flow manifests as SQL queries, which are externally observable in the database traffic. Programs not in the Konure DSL may perform calculations, such as arithmetics and string manipulations, using general-purpose programming language features that are not observable by Konure. These calculations may produce values that do not equal any of the inputs or database values. In this case Konure detects the unanticipated value, reports an error, and exits prematurely.

Example 21. Consider the following Python program inspired by the applications in our experiments:

```
def outside(conn, inputs):
    x = average(inputs[0], inputs[1])
    s1 = util.do_sql(conn, 'SELECT * FROM t1 WHERE val = :x', {'x': x})
    print(s1)
```

The database has a table `t1` with two columns, `id` and `val`, both holding integers. The `conn` variable is an established database connection. The `inputs` variable holds the list of input parameters. This program uses two input parameters, inputs[0] and inputs[1], both assumed to be integers. The program first calculates the average value of the two input parameters and stores it in variable x. Note that this calculation is not expressible in the Konure DSL. Also, the value of x may not equal any of the inputs or database values. The program then calls util.do_sql to perform an SQL query using the value of x.

When inferring this program, our current Konure implementation often reports that the query contains an unanticipated value for which Konure cannot find an origin location. This behavior triggers an assertion failure in our current Konure implementation.

Potential Non-termination: There are adversarial programs for which Konure might not terminate, nor report an error.

Example 22. Consider the following adversarial program, written in Python:

```
def outside(conn, inputs):
  v = inputs[0]
  while True:
    s1 = util.do_sql(conn, 'SELECT * FROM t1 WHERE id = :x', {'x': v})
    if util.has_rows(s1):
      print(v)
      v = util.get_one_data(s1, 't1', 'val')
    else:
      break
  print('Done')
```

The database has a table t1 with two columns, id and val, both holding integers. The column id is the unique primary key. The conn variable is an established database connection. The inputs variable holds the list of input parameters. This program uses one input parameter, inputs[0]. The call to util.do_sql first assembles an SQL query by replacing ":x" with the value of variable v, then performs this query on the database, and finally stores the retrieved rows in variable s1. Because this query selects rows by the primary key, the query always retrieves at most one row. The call to util.has_rows checks whether variable s1 holds nonempty rows. When s1 holds nonempty rows, which must be exactly one row in this program, the call to util.get_one_data extracts from this row the integer in column val. The program then uses the extracted value to update variable v.

If we use Konure to infer this program as a black box, the inference algorithm may not terminate. Recall that the inference algorithm repeatedly represents an unvisited branch as a path constraint and uses this path constraint to solve for a satisfying context. It is always possible for the solver to return a context that causes Konure to infer that the program contains deeper nested conditional branches. For example, let variable i be the input parameter and queries Q_k be as follows ($k = 1, 2, 3, \ldots$):

$$Q_1 = y_1 \leftarrow \textbf{select } \texttt{t1.id}, \texttt{t1.val} \textbf{ where } \texttt{t1.id} = i\,; \textbf{print } [\texttt{t1.id}],$$
$$Q_{k+1} = y_{k+1} \leftarrow \textbf{select } \texttt{t1.id}, \texttt{t1.val} \textbf{ where } \texttt{t1.id} = y_k.\texttt{t1.val}\,; \textbf{print } [\texttt{t1.id}].$$

For each $k = 1, 2, 3, \ldots$, the path constraint

$$W_k = \langle Q_1, \geq 1, \texttt{true}\rangle, \ldots \langle Q_k, \geq 1, \texttt{true}\rangle$$

always has a satisfying context that allows the program above to terminate when executed. If the solver for Konure returns these contexts, the inference algorithm could update the hypothesis, P, as follows:

$$P = \texttt{if}\, Q_1 \,\texttt{then}\, P_1 \,\texttt{else}\, \epsilon,$$
$$P = \texttt{if}\, Q_1 \,\texttt{then}\, \{\, \texttt{if}\, Q_2 \,\texttt{then}\, P_2 \,\texttt{else}\, \epsilon \,\} \,\texttt{else}\, \epsilon,$$
$$P = \texttt{if}\, Q_1 \,\texttt{then}\, \{\, \texttt{if}\, Q_2 \,\texttt{then}\, \{\, \texttt{if}\, Q_3 \,\texttt{then}\, P_3 \,\texttt{else}\, \epsilon \,\} \,\texttt{else}\, \epsilon \,\} \,\texttt{else}\, \epsilon,$$
$$P = \ldots,$$

where P_1, P_2, P_3 denote Prog nonterminals that remain to be inferred. Here, the inference algorithm would populate table `t1` with more and more rows, updating the hypothesis with deeper and deeper nested conditional statements. The hypothesis would always contain an unvisited branch for the case where the last query in the trace retrieves nonempty data. Hence, the inference algorithm would not terminate in this adversarial situation.

Our current Konure implementation uses an off-the-shelf SMT solver that is not maximally distinct. As a result the solver often returns a context that causes the program to enter an infinite loop when executed without allowing our Konure implementation to proceed to non-termination as described above.

6.3 Performance on Synthetic Commands

We evaluate the scalability of the inference algorithm with experiments on the following classes of synthetic commands. The source code for these commands is available in Appendix G and Reference [5].

- **Simple Sequences (SS):** A sequence of different queries, without any conditional or loop statements. Each query does not reference any previously retrieved data.
- **Nested Conditionals (NC):** A series of nested conditional statements. Each except the innermost If statement has a nested If statement in the `then` branch. The innermost If statement has a query in the `then` branch. None of the queries reference previously retrieved data.
- **Unambiguous Long Reference Chains (UL):** Like (NC), but each query references data retrieved by the previous query when the data is nonempty.
- **Ambiguous Long Reference Chains (AL):** Like (UL), but each `then` block has an additional query before the nested If statement. This additional query retrieves a superset of the data that will be retrieved by the next query.
- **Ambiguous Short Reference Chains (AS):** Like (NC), but each `then` block has an additional query before the nested If statement. This additional query retrieves a superset of the data that will be retrieved by the next query, which prints the retrieved data.

We expect the current Konure implementation to (1) scale well for (SS) and (NC) commands—the fact that the queries are independent makes it straightforward to translate path constraints to a small number of logical formulas, (2) scale well for (UL) commands, because disambiguation is unnecessary, (3) scale poorly for (AL) commands, because the number of disambiguation constraints grows rapidly as the length of the query reference chain increases, and (4) scale well for (AS) commands, because the reference chains are short.

For each class above, we built representative commands with varying code sizes. We then used Konure to infer each command. Figure 21 presents statistics from running Konure on these synthetic commands. For SS commands (Figure 21(a)), the horizontal axis presents the number of queries in the command. For the remaining commands (Figures 21(b)–21(e)), the horizontal axis presents the number of conditionals in the command plus one. The left vertical axis presents the number of runs, solves, or lines of code. The lines **Runs** (executions of the command), **Solves**

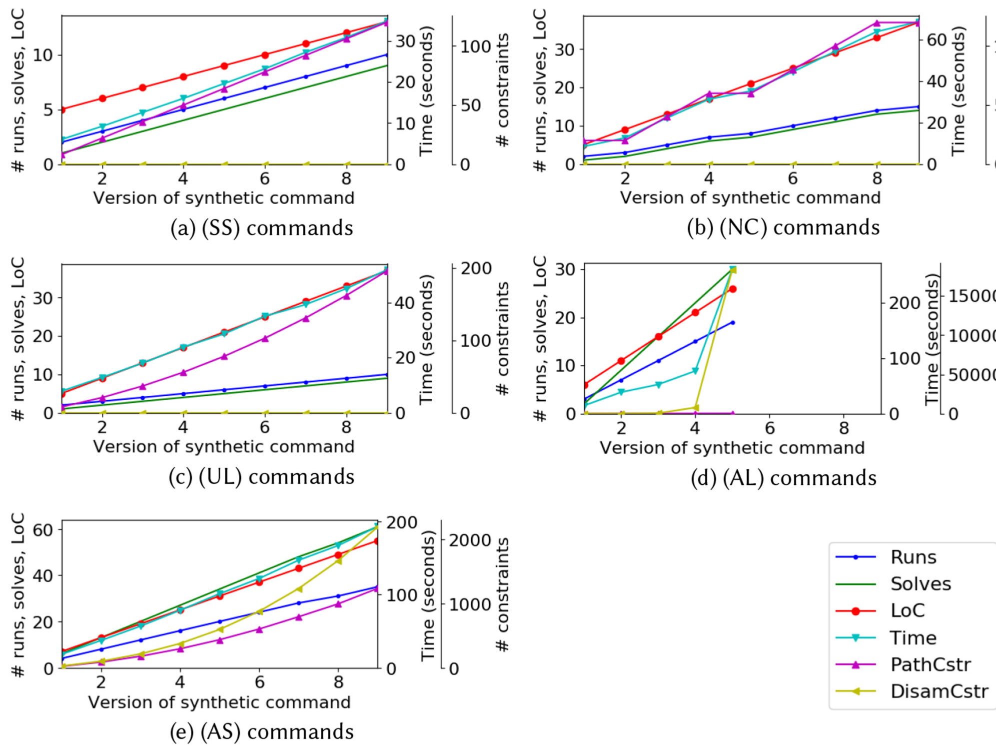

Fig. 21. Performance on synthetic commands.

(invocations of Z3), and **LoC** (lines of code in the command) use this axis. The first right vertical axis presents the inference time in seconds. The line **Time** (wall-clock time for inference) uses this axis. The second right vertical axis presents the number of constraints that Konure sends to the SMT solver during inference. The lines **PathCstr** (constraints to enforce an execution path) and **DisamCstr** (constraints to disambiguate origin locations) use this axis. In Figure 21(d), Konure ran out of memory after the version with five conditionals.

6.3.1 Discussion. Konure scales well for (SS), (NC), (UL), and (AS) commands, which is consistent with results in Section 6.1. Konure does not scale well for (AL) commands, where the major performance bottleneck is sending the solver disambiguation constraints (Section 3.4). We did not optimize Konure to generate a small number of disambiguation constraints, so the communication dominates the inference time. After Z3 receives constraints, it solves them quickly.

We anticipate that commands with ambiguous long reference chains will occur rarely in practice, as the structure of database tables typically supports the application functionality well enough to access the desired data by navigating through only several tables. The four commands from Table 1 with the longest inference times (`get_projects_id`, `get_projects_id_users`, `get_channels`, and `get_channels_id_activities`) all infer in feasible times. We therefore anticipate the inference algorithm will scale to handle real applications.

Since we expect ambiguous long reference chains to occur rarely, we did not optimize Konure for this case. If this issue becomes important in practice, a way to mitigate it would be to develop a solver that returns maximally distinct values. This solver would ensure that unrelated origin locations hold disjoint values.

Because Konure analyzes each command separately, it scales linearly with the number of commands. Therefore, it easily scales to handle applications with many commands, which is often the primary source of complexity.

7 RELATED WORK

A prior version of this research appears in PLDI 2019 [71]. This article adds a full proof for the theorems, along with many definitions that the PLDI version omitted.

Active Learning: Active learning is a classical topic in machine learning [69]. Our approach is characterized by its extensive exploitation of structure present in the program inference task: (1) learning outcomes specified by a DSL, (2) hypotheses as sentential forms in the DSL, and (3) learning by resolving nonterminals in the current hypothesis.

We next discuss related active learning techniques in programming language research, especially for inferring program models.

Our previous research produced an active learning technique for black-box inference of programs that manipulate key/value maps [67]. Konure, in contrast, also observes database traffic, works with a broader and more expressive class of applications, and deploys a top-down, syntax-guided inference algorithm (as opposed to enumerating store/retrieve pairs as in Reference [67]). Our previous research also produced an active learning technique that infers in-memory data structure accesses in certain Python programs, models these accesses with database queries, and uses database implementations instead of the in-memory data structures to regenerate the programs [22, 85]. Konure, in contrast, observes the use of an existing external database, works with programs implemented in any programming language, and guarantees sound and complete inference for programs in the Konure DSL (as opposed to providing probabilistic correctness properties as in References [22, 85]).

Brahma implements oracle-guided synthesis for loop-free programs that compute functions of finite-precision bit-vector inputs [51]. Brahma finitizes the synthesis problem by working with a finite set of components, with each component used exactly once in the synthesized model. Konure, in contrast, works with an infinite space of models with nested control flow.

Mimic traces memory accesses to synthesize a model of a traced function [47]. It uses a random generate-and-test search over a space of programs generated by code mutation operators. There is no guarantee that the generated model is correct or that the search will find a model if one exists.

ALPS uses active learning to prune the search space for synthesizing Datalog programs, which consist of rules [72]. Konure, in contrast, works with database programs that contain database queries, value references, and nested control flow.

Other related techniques include an active learning technique for learning commutativity specifications of data structures [39], a technique for learning program input grammars [15], a technique for learning points-to specifications [16], a technique for learning models of the design patterns that Java computations implement [49], a technique for learning classifiers for event-transition behavior [19], and a technique for inferring the input parsing functionality of programs [22]. Unlike Konure, all of these techniques focus on characterizing specific aspects of program behavior and do not aspire to capture the complete behavior of the application.

Other areas of programming language research have also used active learning, such as for ranking relevant code [82], ranking anomaly reports [55], and improving candidate assertions [60].

Program Synthesis: The vast majority of program synthesis research works with a given set of input/output examples [10, 17, 32–35, 43, 50, 61, 62, 74, 80, 83, 87–89]. Because the examples typically underspecify the program behavior, there are often many programs that satisfy the examples. The synthesized program is therefore typically selected according to either the choices the solver makes [50] or a heuristic that ranks synthesized programs (for example, ranking shorter programs above longer programs) [32, 35, 43]. Konure, in contrast, uses active learning to choose inputs and database contents that eliminate uncertainty and obtain a model that completely captures the core application functionality.

SyGuS identifies a range of program synthesis problems for which it is productive to structure the search space as a DSL [10]. Unlike SyGuS, Konure deploys a top-down inference algorithm that progressively refines a working hypothesis represented as a sentential form of the DSL grammar. Unlike the vast majority of solver-driven synthesis algorithms (which require finite search spaces), Konure works effectively with an unbounded space of models.

LaSy works with a sequence of user-provided input/output pairs to iteratively generalize an overspecialized program [58]. Konure, in contrast, (1) automatically generates a sequence of inputs and database contents that uniquely identify the program within the DSL, (2) observes not just inputs and outputs, but also the traffic between the database and the application, and (3) uses a top-down approach that iteratively resolves DSL grammar nonterminals as opposed to a bottom-up approach that replaces overspecialized code fragments.

Reference [14] presents a static technique that rewrites source code to optimize the execution of loops. Konure, in contrast, does not work with the source code and uses active learning over program executions to infer the program behavior.

To better evaluate the value of active learning in our context, we implemented a system that observes inputs, outputs, and database traffic generated during normal use to infer models of programs that access databases [70]. The results show that this approach often fails to infer the full functionality of the application, because it often misses infrequent corner cases. In contrast, Konure uses active learning to find inputs, as opposed to asking the user for examples or specifications. Wrapping a standard CEGIS-style loop [74] around this system would require access to a specification, such as the source code of a reference implementation, that describes the program behavior to synthesize. In contrast, Konure treats the given program as a black box and infers the program behavior based on its externally visible inputs, outputs, and database traffic.

State Machine Model Learning: State machine learning algorithms [8, 11, 23, 25, 36, 42, 48, 56, 64, 76, 79] construct partial representations of program functionality in the form of finite automata with states and transition rules. State fuzzing tools [7, 31, 63] hypothesize state machines for programs. Network function state model extraction [86] uses program slicing and models the sliced partial programs as packet-processing automata. Konure, in contrast, infers complete application functionality (as opposed to a partial model of the application) and can support application regeneration.

Dynamic Analysis for Program Comprehension: There is a large body of research on dynamic analysis for program comprehension, but (due to complicated logic of Web technologies) relatively little of this research targets Web application servers [29]. WAFA [9] analyzes Web applications, focusing on interactions between Web components, using source code annotations. In contrast, Konure infers applications without analyzing, modifying, or requiring access to source code. Konure works for applications written in any language and can infer both Web and non-Web applications that interact with an external relational database.

DAViS [57] visualizes the data-manipulation behavior of an execution of a data-intensive program. DAViS detects loops whose body contains only one query. DiscoTect [91] summarizes the software architecture of a running object-oriented system as a state machine. They both analyze program behavior when processing certain user-specified inputs. In contrast, KONURE actively explores the execution paths of the program by solving for inputs and database contents that enable it to infer the application behavior.

Database Reverse Engineering/Reengineering: Database reverse engineering analyzes a program's data access patterns, often to reconstruct implicit assumptions of the database schema [27, 30]. KONURE infers programs that interact with databases (and not the structure of the database).

Database program reengineering often involves analyzing the source code to produce more efficient database queries [24, 28]. In contrast, KONURE (1) does not require dynamic program instrumentation or static analysis, (2) does not require the program to be written in specific languages or patterns, and (3) regenerates a new executable program (instead of transforming database queries).

Input Generation for Discovering Defects: Concolic testing [21, 40, 41, 68] generates inputs that systematically explore all execution paths in the program. The goal is to find inputs that expose software defects. BuzzFuzz [38] generates inputs that target defects that occur because of coding oversights at the boundary between application and library code. DIODE [73] generates inputs that target integer overflow errors. All of these techniques target programs written in general-purpose languages such as C. Given the complexity and generality of computations as expressed in this form, completely exploring and characterizing application behavior is infeasible in this context. Our approach, in contrast, (1) works with applications whose behavior can be productively modeled with programs in our DSL and (2) infers a model that captures the complete functionality of the program.

8 CONCLUSION

Applications that read relational databases are pervasive in modern computing environments. We present new active learning techniques that automatically infer and regenerate these applications. Key aspects of these techniques include (1) the formulation of an inferrable DSL that supports the range of computational patterns that these applications exhibit and (2) the inference algorithm, which progressively synthesizes inputs and database contents that productively resolve uncertainty in the current working hypothesis. Results from our implementation highlight the ability of this approach to infer and regenerate applications that access relational databases.

Looking towards the future, we see opportunities extending these techniques. An immediate extension would be expanding the DSL with domain-specific knowledge that enables more effective generation of inputs and database contents. More broadly, future work might expand the domains of computations that work with active learning and identify other crucial components of complex systems that may benefit from inference and regeneration. Another future direction would be to intervene, in addition to observing, the application behavior during execution. A goal here would be to leverage the intervention to more effectively expose learnable application behavior.

APPENDICES

A DEFINITIONS

Figure 22 presents the syntax for skeleton programs (Definition 3.1). We write $\mathcal{S}$ for the set of skeleton programs, $\mathcal{S}$ = SProg. Clearly, for any program $P \in$ Prog, query $Q \in$ Query, and expression $E \in$ Expr, we have $\pi_{\mathcal{S}} P \in \mathcal{S}$, $\pi_{\mathcal{S}} Q \in$ SQuery, and $\pi_{\mathcal{S}} E \in$ SExpr.

$$
\begin{array}{lcl}
\text{SProg} & := & \epsilon \mid \text{SSeq} \mid \text{SIf} \mid \text{SFor} \\
\text{SSeq} & := & \text{SQuery SProg} \\
\text{SIf} & := & \texttt{if}\ \text{SQuery}\ \texttt{then}\ \text{SProg}\ \texttt{else}\ \text{SProg} \\
\text{SFor} & := & \texttt{for}\ \text{SQuery}\ \texttt{do}\ \text{SProg}\ \texttt{else}\ \text{SProg} \\
\text{SQuery} & := & \diamond \leftarrow \texttt{select}\ \text{SCol}^{+}\ \texttt{where}\ \text{SExpr}\,;\ \texttt{print}\ \diamond \\
\text{SExpr} & := & \texttt{true} \mid \text{SExpr} \wedge \text{SExpr} \mid \text{SCol} = \text{SCol} \mid \text{SCol} = \diamond \\
\text{SCol} & := & t\,.\,c
\end{array}
$$

$$t \in Table, \quad c \in Column, \quad \diamond \text{ is a placeholder}$$

Fig. 22. Grammar for skeleton programs ($\mathcal{S}$).

B REGENERATED CODE FOR FULCRUM TASK MANAGER

B.1 Fulcrum Task Manager Command `get_home`

```
def get_home (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'email'))
    if util.has_rows(s0):
        s7 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s7, 'users', 'email'))
        s8 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s7, 'users', 'id')})
        outputs.extend(util.get_data(s8, 'projects', 'id'))
        outputs.extend(util.get_data(s8, 'projects', 'name'))
        outputs.extend(util.get_data(s8, 'projects', 'start_date'))
        s9 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s7, 'users', 'id')})
        outputs.extend(util.get_data(s9, 'users', 'email'))
        s10 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s9, 'users', 'id')})
        outputs.extend(util.get_data(s10, 'projects', 'id'))
        outputs.extend(util.get_data(s10, 'projects', 'name'))
        outputs.extend(util.get_data(s10, 'projects', 'start_date'))
    else:
        pass
    return util.add_warnings(outputs)
```

B.2 Fulcrum Task Manager Command get_projects

```
def get_projects (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'email'))
    if util.has_rows(s0):
        s7 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s7, 'users', 'email'))
        s8 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s7, 'users', 'id')})
        outputs.extend(util.get_data(s8, 'projects', 'id'))
        outputs.extend(util.get_data(s8, 'projects', 'name'))
        outputs.extend(util.get_data(s8, 'projects', 'start_date'))
        s9 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s7, 'users', 'id')})
        outputs.extend(util.get_data(s9, 'users', 'email'))
        s10 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s9, 'users', 'id')})
        outputs.extend(util.get_data(s10, 'projects', 'id'))
        outputs.extend(util.get_data(s10, 'projects', 'name'))
        outputs.extend(util.get_data(s10, 'projects', 'start_date'))
    else:
        pass
    return util.add_warnings(outputs)
```

B.3 Fulcrum Task Manager Command get_projects_id

```
def get_projects_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s9 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s10 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s11 = util.do_sql(conn, "SELECT  DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0 AND `projects`.`
            id` = :x1 LIMIT 1", {'x0': util.get_one_data(s10, 'users', 'id'), '
            x1': inputs[1]})
        outputs.extend(util.get_data(s11, 'projects', 'id'))
        outputs.extend(util.get_data(s11, 'projects', 'name'))
        if util.has_rows(s11):
            s61 = util.do_sql(conn, "SELECT DISTINCT `users`.* FROM `users`
                INNER JOIN `projects_users` ON `users`.`id` = `projects_users`.`
                user_id` WHERE `projects_users`.`project_id` = :x0", {'x0': util
                .get_one_data(s11, 'projects', 'id')})
            outputs.extend(util.get_data(s61, 'users', 'id'))
            outputs.extend(util.get_data(s61, 'users', 'email'))
            outputs.extend(util.get_data(s61, 'users', 'name'))
            outputs.extend(util.get_data(s61, 'users', 'initials'))
            s62 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
            outputs.extend(util.get_data(s62, 'projects', 'id'))
            outputs.extend(util.get_data(s62, 'projects', 'name'))
        else:
            s12 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

B.4 Fulcrum Task Manager Command get_projects_id_stories

```
def get_projects_id_stories (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s9 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s10 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s11 = util.do_sql(conn, "SELECT  DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0 AND `projects`.`
            id` = :x1 LIMIT 1", {'x0': util.get_one_data(s10, 'users', 'id'), '
            x1': inputs[1]})
        if util.has_rows(s11):
            s46 = util.do_sql(conn, "SELECT `stories`.* FROM `stories`  WHERE `
                stories`.`project_id` IN (:x0)", {'x0': util.get_one_data(s11, '
                projects', 'id')})
            outputs.extend(util.get_data(s46, 'stories', 'id'))
            outputs.extend(util.get_data(s46, 'stories', 'title'))
            outputs.extend(util.get_data(s46, 'stories', 'description'))
            outputs.extend(util.get_data(s46, 'stories', 'estimate'))
            outputs.extend(util.get_data(s46, 'stories', 'requested_by_id'))
            outputs.extend(util.get_data(s46, 'stories', 'owned_by_id'))
            outputs.extend(util.get_data(s46, 'stories', 'project_id'))
            if util.has_rows(s46):
                s62 = util.do_sql(conn, "SELECT `notes`.* FROM `notes`  WHERE `
                    notes`.`story_id` IN (:x0)", {'x0': util.get_one_data(s46, '
                    stories', 'id')})
                outputs.extend(util.get_data(s62, 'notes', 'id'))
                outputs.extend(util.get_data(s62, 'notes', 'note'))
                outputs.extend(util.get_data(s62, 'notes', 'user_id'))
                outputs.extend(util.get_data(s62, 'notes', 'story_id'))
            else:
                pass
        else:
            s12 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

B.5 Fulcrum Task Manager Command get_projects_id_stories_id

```
def get_projects_id_stories_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s9 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s10 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s11 = util.do_sql(conn, "SELECT  DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0 AND `projects`.`
            id` = :x1 LIMIT 1", {'x0': util.get_one_data(s10, 'users', 'id'), '
            x1': inputs[1]})
        if util.has_rows(s11):
            s47 = util.do_sql(conn, "SELECT  `stories`.* FROM `stories`  WHERE `
                stories`.`project_id` = :x0 AND `stories`.`id` = :x1 LIMIT 1", {
                'x0': util.get_one_data(s11, 'projects', 'id'), 'x1': inputs
                [2]})
            outputs.extend(util.get_data(s47, 'stories', 'id'))
            outputs.extend(util.get_data(s47, 'stories', 'title'))
            outputs.extend(util.get_data(s47, 'stories', 'description'))
            outputs.extend(util.get_data(s47, 'stories', 'estimate'))
            outputs.extend(util.get_data(s47, 'stories', 'requested_by_id'))
            outputs.extend(util.get_data(s47, 'stories', 'owned_by_id'))
            outputs.extend(util.get_data(s47, 'stories', 'project_id'))
            if util.has_rows(s47):
                s64 = util.do_sql(conn, "SELECT `notes`.* FROM `notes`  WHERE `
                    notes`.`story_id` = :x0", {'x0': util.get_one_data(s47, '
                    stories', 'id')})
                outputs.extend(util.get_data(s64, 'notes', 'id'))
                outputs.extend(util.get_data(s64, 'notes', 'note'))
                outputs.extend(util.get_data(s64, 'notes', 'user_id'))
                outputs.extend(util.get_data(s64, 'notes', 'story_id'))
            else:
                s48 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `
                    projects` INNER JOIN `projects_users` ON `projects`.`id` = `
                    projects_users`.`project_id` WHERE `projects_users`.`user_id
                    ` = :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
        else:
            s12 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

B.6 Fulcrum Task Manager Command get_projects_id_stories_id_notes

```
def get_projects_id_stories_id_notes (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s9 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s10 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s11 = util.do_sql(conn, "SELECT  DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0 AND `projects`.`
            id` = :x1 LIMIT 1", {'x0': util.get_one_data(s10, 'users', 'id'), '
            x1': inputs[1]})
        if util.has_rows(s11):
            s47 = util.do_sql(conn, "SELECT  `stories`.* FROM `stories`  WHERE `
                stories`.`project_id` = :x0 AND `stories`.`id` = :x1 LIMIT 1", {
                'x0': util.get_one_data(s11, 'projects', 'id'), 'x1': inputs
                [2]})
            if util.has_rows(s47):
                s57 = util.do_sql(conn, "SELECT `notes`.* FROM `notes`  WHERE `
                    notes`.`story_id` = :x0", {'x0': util.get_one_data(s47, '
                    stories', 'id')})
                outputs.extend(util.get_data(s57, 'notes', 'id'))
                outputs.extend(util.get_data(s57, 'notes', 'note'))
                outputs.extend(util.get_data(s57, 'notes', 'user_id'))
                outputs.extend(util.get_data(s57, 'notes', 'story_id'))
            else:
                s48 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `
                    projects` INNER JOIN `projects_users` ON `projects`.`id` = `
                    projects_users`.`project_id` WHERE `projects_users`.`user_id
                    ` = :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
        else:
            s12 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

B.7 Fulcrum Task Manager Command `get_projects_id_stories_id_notes_id`

```
def get_projects_id_stories_id_notes_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s9 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s10 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s11 = util.do_sql(conn, "SELECT  DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0 AND `projects`.`
            id` = :x1 LIMIT 1", {'x0': util.get_one_data(s10, 'users', 'id'), '
            x1': inputs[1]})
        if util.has_rows(s11):
            s47 = util.do_sql(conn, "SELECT  `stories`.* FROM `stories`  WHERE `
                stories`.`project_id` = :x0 AND `stories`.`id` = :x1 LIMIT 1", {
                'x0': util.get_one_data(s11, 'projects', 'id'), 'x1': inputs
                [2]})
            if util.has_rows(s47):
                s57 = util.do_sql(conn, "SELECT  `notes`.* FROM `notes`  WHERE `
                    notes`.`story_id` = :x0 AND `notes`.`id` = :x1 LIMIT 1", {'
                    x0': util.get_one_data(s47, 'stories', 'id'), 'x1': inputs
                    [3]})
                outputs.extend(util.get_data(s57, 'notes', 'id'))
                outputs.extend(util.get_data(s57, 'notes', 'note'))
                outputs.extend(util.get_data(s57, 'notes', 'user_id'))
                outputs.extend(util.get_data(s57, 'notes', 'story_id'))
                if util.has_rows(s57):
                    pass
                else:
                    s58 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `
                        projects` INNER JOIN `projects_users` ON `projects`.`id`
                         = `projects_users`.`project_id` WHERE `projects_users
                        `.`user_id` = :x0", {'x0': util.get_one_data(s10, 'users
                        ', 'id')})
            else:
                s48 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `
                    projects` INNER JOIN `projects_users` ON `projects`.`id` = `
                    projects_users`.`project_id` WHERE `projects_users`.`user_id
                    ` = :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
        else:
            s12 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

B.8 Fulcrum Task Manager Command get_projects_id_users

```
def get_projects_id_users (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`email
        ` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s9 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s10 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0  ORDER BY `users`.`id` ASC LIMIT 1", {'x0': util.
            get_one_data(s8, 'users', 'id')})
        s11 = util.do_sql(conn, "SELECT  DISTINCT `projects`.* FROM `projects`
            INNER JOIN `projects_users` ON `projects`.`id` = `projects_users`.`
            project_id` WHERE `projects_users`.`user_id` = :x0 AND `projects`.`
            id` = :x1 LIMIT 1", {'x0': util.get_one_data(s10, 'users', 'id'), '
            x1': inputs[1]})
        outputs.extend(util.get_data(s11, 'projects', 'id'))
        outputs.extend(util.get_data(s11, 'projects', 'name'))
        if util.has_rows(s11):
            s61 = util.do_sql(conn, "SELECT DISTINCT `users`.* FROM `users`
                INNER JOIN `projects_users` ON `users`.`id` = `projects_users`.`
                user_id` WHERE `projects_users`.`project_id` = :x0", {'x0': util
                .get_one_data(s11, 'projects', 'id')})
            outputs.extend(util.get_data(s61, 'users', 'id'))
            outputs.extend(util.get_data(s61, 'users', 'email'))
            outputs.extend(util.get_data(s61, 'users', 'name'))
            outputs.extend(util.get_data(s61, 'users', 'initials'))
            s62 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
            outputs.extend(util.get_data(s62, 'projects', 'id'))
            outputs.extend(util.get_data(s62, 'projects', 'name'))
        else:
            s12 = util.do_sql(conn, "SELECT DISTINCT `projects`.* FROM `projects
                ` INNER JOIN `projects_users` ON `projects`.`id` = `
                projects_users`.`project_id` WHERE `projects_users`.`user_id` =
                :x0", {'x0': util.get_one_data(s10, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

C REGENERATED CODE FOR KANDAN CHAT ROOM

C.1 Kandan Chat Room Command get_channels

```
def get_channels (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        outputs.extend(util.get_data(s3, 'channels', 'id'))
        outputs.extend(util.get_data(s3, 'channels', 'name'))
        outputs.extend(util.get_data(s3, 'channels', 'user_id'))
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3
                , 'channels', 'id')})
            outputs.extend(util.get_data(s4, 'activities', 'id'))
            outputs.extend(util.get_data(s4, 'activities', 'content'))
            outputs.extend(util.get_data(s4, 'activities', 'channel_id'))
            outputs.extend(util.get_data(s4, 'activities', 'user_id'))
            if util.has_rows(s4):
                s71 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `
                    users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities',
                    'user_id')})
                outputs.extend(util.get_data(s71, 'users', 'id'))
                outputs.extend(util.get_data(s71, 'users', 'email'))
                outputs.extend(util.get_data(s71, 'users', 'first_name'))
                outputs.extend(util.get_data(s71, 'users', 'last_name'))
                outputs.extend(util.get_data(s71, 'users', 'username'))
                s72 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                s73 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`",
                    {})
                outputs.extend(util.get_data(s73, 'channels', 'id'))
                outputs.extend(util.get_data(s73, 'channels', 'name'))
                outputs.extend(util.get_data(s73, 'channels', 'user_id'))
                s73_all = s73
                for s73 in s73_all:
                    s74 = util.do_sql(conn, "SELECT COUNT(*) FROM `activities`
                        WHERE `activities`.`channel_id` = :x0", {'x0': util.
                        get_one_data(s73, 'channels', 'id')})
                    s75 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0 ORDER
```

```
                     BY id DESC LIMIT 30 OFFSET 0", {'x0': util.get_one_data
                    (s73, 'channels', 'id')})
                outputs.extend(util.get_data(s75, 'activities', 'id'))
                outputs.extend(util.get_data(s75, 'activities', 'content'))
                outputs.extend(util.get_data(s75, 'activities', 'channel_id'
                    ))
                outputs.extend(util.get_data(s75, 'activities', 'user_id'))
                if util.has_rows(s75):
                    s78 = util.do_sql(conn, "SELECT `users`.* FROM `users`
                        WHERE `users`.`id` IN :x0", {'x0': util.get_data(s75
                        , 'activities', 'user_id')})
                    outputs.extend(util.get_data(s78, 'users', 'id'))
                    outputs.extend(util.get_data(s78, 'users', 'email'))
                    outputs.extend(util.get_data(s78, 'users', 'first_name')
                        )
                    outputs.extend(util.get_data(s78, 'users', 'last_name'))
                    outputs.extend(util.get_data(s78, 'users', 'username'))
                else:
                    pass
            s73 = s73_all
        else:
            s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                users', 'id')})
            s6 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`",
                {})
            outputs.extend(util.get_data(s6, 'channels', 'id'))
            outputs.extend(util.get_data(s6, 'channels', 'name'))
            outputs.extend(util.get_data(s6, 'channels', 'user_id'))
            s6_all = s6
            for s6 in s6_all:
                s7 = util.do_sql(conn, "SELECT COUNT(*) FROM `activities`
                    WHERE `activities`.`channel_id` = :x0", {'x0': util.
                    get_one_data(s6, 'channels', 'id')})
                s8 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                    activities`  WHERE `activities`.`channel_id` = :x0 ORDER
                     BY id DESC LIMIT 30 OFFSET 0", {'x0': util.get_one_data
                    (s6, 'channels', 'id')})
            s6 = s6_all
    else:
        s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
            users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users'
            , 'id')})
        s37 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
else:
    pass
return util.add_warnings(outputs)
```

C.2 Kandan Chat Room Command get_channels_id_activities

```
def get_channels_id_activities (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3
                , 'channels', 'id')})
            if util.has_rows(s4):
                s40 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `
                    users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities',
                    'user_id')})
                s41 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                s42 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels`
                    WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
                if util.has_rows(s42):
                    s132 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0 ORDER
                         BY id LIMIT 1", {'x0': util.get_one_data(s42, 'channels
                        ', 'id')})
                    outputs.extend(util.get_data(s132, 'activities', 'id'))
                    outputs.extend(util.get_data(s132, 'activities', 'content'))
                    outputs.extend(util.get_data(s132, 'activities', 'channel_id
                        '))
                    outputs.extend(util.get_data(s132, 'activities', 'user_id'))
                    s133 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                        activities`  WHERE `activities`.`channel_id` = :x0 ORDER
                         BY id DESC LIMIT 30", {'x0': util.get_one_data(s42, '
                        channels', 'id')})
                    outputs.extend(util.get_data(s133, 'activities', 'id'))
                    outputs.extend(util.get_data(s133, 'activities', 'content'))
                    outputs.extend(util.get_data(s133, 'activities', 'channel_id
                        '))
```

```
                outputs.extend(util.get_data(s133, 'activities', 'user_id'))
                if util.has_rows(s133):
                    s167 = util.do_sql(conn, "SELECT `users`.* FROM `users`
                         WHERE `users`.`id` IN :x0", {'x0': util.get_data(
                        s133, 'activities', 'user_id')})
                    outputs.extend(util.get_data(s167, 'users', 'id'))
                    outputs.extend(util.get_data(s167, 'users', 'email'))
                    outputs.extend(util.get_data(s167, 'users', 'first_name'
                        ))
                    outputs.extend(util.get_data(s167, 'users', 'last_name')
                        )
                    outputs.extend(util.get_data(s167, 'users', 'username'))
                else:
                    pass
            else:
                pass
        else:
            s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                users', 'id')})
            s6 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels`
                WHERE `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
            if util.has_rows(s6):
                s69 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                    activities`  WHERE `activities`.`channel_id` = :x0 ORDER
                     BY id LIMIT 1", {'x0': util.get_one_data(s6, 'channels'
                    , 'id')})
                s70 = util.do_sql(conn, "SELECT  `activities`.* FROM `
                    activities`  WHERE `activities`.`channel_id` = :x0 ORDER
                     BY id DESC LIMIT 30", {'x0': util.get_one_data(s6, '
                    channels', 'id')})
            else:
                pass
    else:
        s25 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
            users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users'
            , 'id')})
        s26 = util.do_sql(conn, "SELECT  `channels`.* FROM `channels`  WHERE
             `channels`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
else:
    pass
return util.add_warnings(outputs)
```

C.3 Kandan Chat Room Command get_channels_id_activities_id

```
def get_channels_id_activities_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3
                , 'channels', 'id')})
            if util.has_rows(s4):
                s47 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `
                    users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities',
                    'user_id')})
                s48 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                s49 = util.do_sql(conn, "SELECT  `activities`.* FROM `activities
                    `  WHERE `activities`.`id` = :x0 LIMIT 1", {'x0': inputs
                    [2]})
                outputs.extend(util.get_data(s49, 'activities', 'content'))
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                s6 = util.do_sql(conn, "SELECT  `activities`.* FROM `activities`
                      WHERE `activities`.`id` = :x0 LIMIT 1", {'x0': inputs[2]})
                outputs.extend(util.get_data(s6, 'activities', 'content'))
        else:
            s25 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users'
                , 'id')})
            s26 = util.do_sql(conn, "SELECT  `activities`.* FROM `activities`
                WHERE `activities`.`id` = :x0 LIMIT 1", {'x0': inputs[2]})
            outputs.extend(util.get_data(s26, 'activities', 'content'))
    else:
        pass
    return util.add_warnings(outputs)
```

C.4 Kandan Chat Room Command get_me

```
def get_me (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'first_name'))
    outputs.extend(util.get_data(s0, 'users', 'last_name'))
    outputs.extend(util.get_data(s0, 'users', 'username'))
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s2, 'users', 'id'))
        outputs.extend(util.get_data(s2, 'users', 'email'))
        outputs.extend(util.get_data(s2, 'users', 'first_name'))
        outputs.extend(util.get_data(s2, 'users', 'last_name'))
        outputs.extend(util.get_data(s2, 'users', 'username'))
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3
                , 'channels', 'id')})
            if util.has_rows(s4):
                s35 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `
                    users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities',
                    'user_id')})
                s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                outputs.extend(util.get_data(s36, 'users', 'id'))
                outputs.extend(util.get_data(s36, 'users', 'email'))
                outputs.extend(util.get_data(s36, 'users', 'first_name'))
                outputs.extend(util.get_data(s36, 'users', 'last_name'))
                outputs.extend(util.get_data(s36, 'users', 'username'))
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                outputs.extend(util.get_data(s5, 'users', 'id'))
                outputs.extend(util.get_data(s5, 'users', 'email'))
                outputs.extend(util.get_data(s5, 'users', 'first_name'))
                outputs.extend(util.get_data(s5, 'users', 'last_name'))
                outputs.extend(util.get_data(s5, 'users', 'username'))
        else:
            s22 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users'
                , 'id')})
            outputs.extend(util.get_data(s22, 'users', 'id'))
            outputs.extend(util.get_data(s22, 'users', 'email'))
            outputs.extend(util.get_data(s22, 'users', 'first_name'))
            outputs.extend(util.get_data(s22, 'users', 'last_name'))
            outputs.extend(util.get_data(s22, 'users', 'username'))
    else:
        pass
    return util.add_warnings(outputs)
```

C.5 Kandan Chat Room Command `get_users`

```
def get_users (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'first_name'))
    outputs.extend(util.get_data(s0, 'users', 'last_name'))
    outputs.extend(util.get_data(s0, 'users', 'username'))
    if util.has_rows(s0):
        s8 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s8, 'users', 'id'))
        outputs.extend(util.get_data(s8, 'users', 'email'))
        outputs.extend(util.get_data(s8, 'users', 'first_name'))
        outputs.extend(util.get_data(s8, 'users', 'last_name'))
        outputs.extend(util.get_data(s8, 'users', 'username'))
        s9 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s9):
            s10 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s9
                , 'channels', 'id')})
            if util.has_rows(s10):
                s58 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `
                    users`.`id` IN :x0", {'x0': util.get_data(s10, 'activities',
                     'user_id')})
                outputs.extend(util.get_data(s58, 'users', 'id'))
                outputs.extend(util.get_data(s58, 'users', 'email'))
                outputs.extend(util.get_data(s58, 'users', 'first_name'))
                outputs.extend(util.get_data(s58, 'users', 'last_name'))
                outputs.extend(util.get_data(s58, 'users', 'username'))
                s59 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s8, '
                    users', 'id')})
                outputs.extend(util.get_data(s59, 'users', 'id'))
                outputs.extend(util.get_data(s59, 'users', 'email'))
                outputs.extend(util.get_data(s59, 'users', 'first_name'))
                outputs.extend(util.get_data(s59, 'users', 'last_name'))
```

```
                outputs.extend(util.get_data(s59, 'users', 'username'))
                s60 = util.do_sql(conn, "SELECT `users`.* FROM `users`", {})
                outputs.extend(util.get_data(s60, 'users', 'id'))
                outputs.extend(util.get_data(s60, 'users', 'email'))
                outputs.extend(util.get_data(s60, 'users', 'first_name'))
                outputs.extend(util.get_data(s60, 'users', 'last_name'))
                outputs.extend(util.get_data(s60, 'users', 'username'))
            else:
                s11 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s8, '
                    users', 'id')})
                outputs.extend(util.get_data(s11, 'users', 'id'))
                outputs.extend(util.get_data(s11, 'users', 'email'))
                outputs.extend(util.get_data(s11, 'users', 'first_name'))
                outputs.extend(util.get_data(s11, 'users', 'last_name'))
                outputs.extend(util.get_data(s11, 'users', 'username'))
                s12 = util.do_sql(conn, "SELECT `users`.* FROM `users`", {})
                outputs.extend(util.get_data(s12, 'users', 'id'))
                outputs.extend(util.get_data(s12, 'users', 'email'))
                outputs.extend(util.get_data(s12, 'users', 'first_name'))
                outputs.extend(util.get_data(s12, 'users', 'last_name'))
                outputs.extend(util.get_data(s12, 'users', 'username'))
        else:
            s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s8, 'users'
                , 'id')})
            outputs.extend(util.get_data(s36, 'users', 'id'))
            outputs.extend(util.get_data(s36, 'users', 'email'))
            outputs.extend(util.get_data(s36, 'users', 'first_name'))
            outputs.extend(util.get_data(s36, 'users', 'last_name'))
            outputs.extend(util.get_data(s36, 'users', 'username'))
            s37 = util.do_sql(conn, "SELECT `users`.* FROM `users`", {})
            outputs.extend(util.get_data(s37, 'users', 'id'))
            outputs.extend(util.get_data(s37, 'users', 'email'))
            outputs.extend(util.get_data(s37, 'users', 'first_name'))
            outputs.extend(util.get_data(s37, 'users', 'last_name'))
            outputs.extend(util.get_data(s37, 'users', 'username'))
    else:
        pass
    return util.add_warnings(outputs)
```

C.6 Kandan Chat Room Command `get_users_id`

```
def get_users_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
        username` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'first_name'))
    outputs.extend(util.get_data(s0, 'users', 'last_name'))
    outputs.extend(util.get_data(s0, 'users', 'username'))
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `users`.`
            id` = :x0 LIMIT 1", {'x0': util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s2, 'users', 'id'))
        outputs.extend(util.get_data(s2, 'users', 'email'))
        outputs.extend(util.get_data(s2, 'users', 'first_name'))
        outputs.extend(util.get_data(s2, 'users', 'last_name'))
        outputs.extend(util.get_data(s2, 'users', 'username'))
        s3 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`", {})
        if util.has_rows(s3):
            s4 = util.do_sql(conn, "SELECT `activities`.* FROM `activities`
                WHERE `activities`.`channel_id` IN :x0", {'x0': util.get_data(s3
                , 'channels', 'id')})
            if util.has_rows(s4):
                s35 = util.do_sql(conn, "SELECT `users`.* FROM `users`  WHERE `
                    users`.`id` IN :x0", {'x0': util.get_data(s4, 'activities',
                    'user_id')})
                s36 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                outputs.extend(util.get_data(s36, 'users', 'id'))
                outputs.extend(util.get_data(s36, 'users', 'email'))
                outputs.extend(util.get_data(s36, 'users', 'first_name'))
                outputs.extend(util.get_data(s36, 'users', 'last_name'))
                outputs.extend(util.get_data(s36, 'users', 'username'))
            else:
                s5 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                    users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, '
                    users', 'id')})
                outputs.extend(util.get_data(s5, 'users', 'id'))
                outputs.extend(util.get_data(s5, 'users', 'email'))
                outputs.extend(util.get_data(s5, 'users', 'first_name'))
                outputs.extend(util.get_data(s5, 'users', 'last_name'))
                outputs.extend(util.get_data(s5, 'users', 'username'))
        else:
            s22 = util.do_sql(conn, "SELECT  `users`.* FROM `users`  WHERE `
                users`.`id` = :x0 LIMIT 1", {'x0': util.get_one_data(s2, 'users'
                , 'id')})
            outputs.extend(util.get_data(s22, 'users', 'id'))
            outputs.extend(util.get_data(s22, 'users', 'email'))
            outputs.extend(util.get_data(s22, 'users', 'first_name'))
            outputs.extend(util.get_data(s22, 'users', 'last_name'))
            outputs.extend(util.get_data(s22, 'users', 'username'))
    else:
        pass
    return util.add_warnings(outputs)
```

D REGENERATED CODE FOR ENKI BLOGGING APPLICATION

D.1 Enki Blogging Application Command get_admin_comments_id

```
def get_admin_comments_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `comments`.* FROM `comments` WHERE `comments
        `.`id` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'comments', 'id'))
    outputs.extend(util.get_data(s0, 'comments', 'author'))
    outputs.extend(util.get_data(s0, 'comments', 'author_url'))
    outputs.extend(util.get_data(s0, 'comments', 'author_email'))
    outputs.extend(util.get_data(s0, 'comments', 'body'))
    return util.add_warnings(outputs)
```

D.2 Enki Blogging Application Command get_admin_pages

```
def get_admin_pages (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT COUNT(*) FROM `pages`", {})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `pages`.* FROM `pages`  ORDER BY
            created_at DESC LIMIT 30 OFFSET 0", {})
        outputs.extend(util.get_data(s2, 'pages', 'id'))
        outputs.extend(util.get_data(s2, 'pages', 'title'))
        outputs.extend(util.get_data(s2, 'pages', 'slug'))
        outputs.extend(util.get_data(s2, 'pages', 'body'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.3 Enki Blogging Application Command get_admin_pages_id

```
def get_admin_pages_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `pages`.* FROM `pages` WHERE `pages`.`id` =
        :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'pages', 'id'))
    outputs.extend(util.get_data(s0, 'pages', 'title'))
    outputs.extend(util.get_data(s0, 'pages', 'slug'))
    outputs.extend(util.get_data(s0, 'pages', 'body'))
    return util.add_warnings(outputs)
```

D.4 Enki Blogging Application Command `get_admin_posts`

```
def get_admin_posts (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT COUNT(*) FROM `posts`", {})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT  `posts`.* FROM `posts`  ORDER BY
            coalesce(published_at, updated_at) DESC LIMIT 30 OFFSET 0", {})
        outputs.extend(util.get_data(s2, 'posts', 'id'))
        outputs.extend(util.get_data(s2, 'posts', 'title'))
        outputs.extend(util.get_data(s2, 'posts', 'body'))
        s2_all = s2
        for s2 in s2_all:
            s3 = util.do_sql(conn, "SELECT COUNT(*) FROM `comments` WHERE `
                comments`.`post_id` = :x0", {'x0': util.get_one_data(s2, 'posts'
                , 'id')})
        s2 = s2_all
    else:
        pass
    return util.add_warnings(outputs)
```

E REGENERATED CODE FOR BLOG APPLICATION

E.1 Blog Application Command `get_articles`

```
def get_articles (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT `articles`.* FROM `articles`", {})
    outputs.extend(util.get_data(s0, 'articles', 'id'))
    outputs.extend(util.get_data(s0, 'articles', 'title'))
    outputs.extend(util.get_data(s0, 'articles', 'text'))
    s1 = util.do_sql(conn, "SELECT `articles`.* FROM `articles`", {})
    outputs.extend(util.get_data(s1, 'articles', 'id'))
    outputs.extend(util.get_data(s1, 'articles', 'title'))
    outputs.extend(util.get_data(s1, 'articles', 'text'))
    return util.add_warnings(outputs)
```

E.2 Blog Application Command get_article_id

```
def get_article_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT `articles`.* FROM `articles`", {})
    s1 = util.do_sql(conn, "SELECT  `articles`.* FROM `articles` WHERE `articles
        `.`id` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s1, 'articles', 'id'))
    outputs.extend(util.get_data(s1, 'articles', 'title'))
    outputs.extend(util.get_data(s1, 'articles', 'text'))
    if util.has_rows(s1):
        s9 = util.do_sql(conn, "SELECT `comments`.* FROM `comments` WHERE `
            comments`.`article_id` = :x0", {'x0': util.get_one_data(s1, '
            articles', 'id')})
        outputs.extend(util.get_data(s9, 'comments', 'commenter'))
        outputs.extend(util.get_data(s9, 'comments', 'body'))
        outputs.extend(util.get_data(s9, 'comments', 'article_id'))
    else:
        pass
    return util.add_warnings(outputs)
```

F REGENERATED CODE FOR STUDENT REGISTRATION SYSTEM

F.1 Student Registration System Command liststudentcourses

```
def liststudentcourses (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM student WHERE id = :x0", {'x0': inputs
        [0]})
    if util.has_rows(s0):
        s2 = util.do_sql(conn, "SELECT * FROM student WHERE id=:x0 AND password
            =:x1", {'x0': util.get_one_data(s0, 'student', 'id'), 'x1': inputs
            [1]})
        if util.has_rows(s2):
            s6 = util.do_sql(conn, "SELECT * FROM course c JOIN registration r
                on r.course_id = c.id WHERE r.student_id = :x0", {'x0': util.
                get_one_data(s2, 'student', 'id')})
            outputs.extend(util.get_data(s6, 'course', 'id'))
            outputs.extend(util.get_data(s6, 'course', 'teacher_id'))
            outputs.extend(util.get_data(s6, 'registration', 'course_id'))
            if util.has_rows(s6):
                s6_all = s6
                for s6 in s6_all:
                    s12 = util.do_sql(conn, "Select firstname, lastname from
                        teacher where id = :x0", {'x0': util.get_one_data(s6, '
                        course', 'teacher_id')})
                    s13 = util.do_sql(conn, "SELECT count(*) FROM registration
                        WHERE course_id = :x0", {'x0': util.get_one_data(s6, '
                        registration', 'course_id')})
                s6 = s6_all
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

G SYNTHETIC COMMANDS

G.1 Simple Sequences (SS)

Version 1:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
```

Version 2:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
```

Version 3:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
```

Version 4:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
    s4 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[4]})
```

Version 5:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
    s4 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[4]})
    s5 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[5]})
```

Version 6:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
    s4 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[4]})
    s5 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[5]})
    s6 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[6]})
```

Version 7:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
    s4 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[4]})
    s5 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[5]})
    s6 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[6]})
    s7 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[7]})
```

Version 8:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
    s4 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[4]})
    s5 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[5]})
    s6 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[6]})
    s7 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[7]})
    s8 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[8]})
```

Version 9:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    s2 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[2]})
    s3 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[3]})
    s4 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[4]})
    s5 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[5]})
    s6 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[6]})
    s7 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[7]})
    s8 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[8]})
    s9 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[9]})
```

G.2 Nested Conditionals (NC)

Version 1:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
```

Version 2:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
```

Version 3:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
```

Version 4:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4", {})
```

Version 5:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4", {})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5", {})
```

Version 6:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4", {})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5", {})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6", {})
```

Version 7:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4", {})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5", {})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6", {})
                        if s6:
                            s7 = util.do_sql(conn, "SELECT * FROM t7", {})
```

Version 8:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4", {})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5", {})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6", {})
                        if s6:
                            s7 = util.do_sql(conn, "SELECT * FROM t7", {})
                            if s7:
                                s8 = util.do_sql(conn, "SELECT * FROM t8", {})
```

Version 9:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1", {})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3", {})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4", {})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5", {})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6", {})
                        if s6:
                            s7 = util.do_sql(conn, "SELECT * FROM t7", {})
                            if s7:
                                s8 = util.do_sql(conn, "SELECT * FROM t8", {})
                                if s8:
                                    s9 = util.do_sql(conn, "SELECT * FROM t9",
                                        {})
```

G.3 Unambiguous Long Reference Chains (UL)

Version 1:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
```

Version 2:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
```

Version 3:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
```

Version 4:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
```

Version 5:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
```

Version 6:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                            {"x": util.get_one_data(s5, 't5', 'val')})
```

Version 7:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": util.get_one_data(s5, 't5', 'val')})
                        if s6:
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": util.get_one_data(s6, 't6', 'val')})
```

Version 8:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": util.get_one_data(s5, 't5', 'val')})
                        if s6:
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": util.get_one_data(s6, 't6', 'val')})
                            if s7:
                                s8 = util.do_sql(conn, "SELECT * FROM t8 WHERE
                                    id = :x", {"x": util.get_one_data(s7, 't7',
                                    'val')})
```

Version 9:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": util.get_one_data(s5, 't5', 'val')})
                        if s6:
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": util.get_one_data(s6, 't6', 'val')})
                            if s7:
                                s8 = util.do_sql(conn, "SELECT * FROM t8 WHERE
                                    id = :x", {"x": util.get_one_data(s7, 't7',
                                    'val')})
                                if s8:
                                    s9 = util.do_sql(conn, "SELECT * FROM t9
                                        WHERE id = :x", {"x": util.get_one_data(
                                        s8, 't8', 'val')})
```

G.4 Ambiguous Long Reference Chains (AL)

Version 1:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
```

Version 2:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
```

Version 3:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
```

Version 4:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
```

Version 5:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
```

Version 6:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                            {"x": util.get_one_data(s5, 't5', 'val')})
```

Version 7:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": util.get_one_data(s5, 't5', 'val')})
                        if s6:
                            s7_ = util.do_sql(conn, "SELECT * FROM t7", {})
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": util.get_one_data(s6, 't6', 'val')})
```

Version 8:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": util.get_one_data(s5, 't5', 'val')})
                        if s6:
                            s7_ = util.do_sql(conn, "SELECT * FROM t7", {})
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": util.get_one_data(s6, 't6', 'val')})
                            if s7:
                                s8_ = util.do_sql(conn, "SELECT * FROM t8", {})
                                s8 = util.do_sql(conn, "SELECT * FROM t8 WHERE
                                    id = :x", {"x": util.get_one_data(s7, 't7',
                                    'val')})
```

Version 9:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": util.
            get_one_data(s1, 't1', 'val')})
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": util.
                get_one_data(s2, 't2', 'val')})
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    util.get_one_data(s3, 't3', 'val')})
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": util.get_one_data(s4, 't4', 'val')})
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": util.get_one_data(s5, 't5', 'val')})
                        if s6:
                            s7_ = util.do_sql(conn, "SELECT * FROM t7", {})
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": util.get_one_data(s6, 't6', 'val')})
                            if s7:
                                s8_ = util.do_sql(conn, "SELECT * FROM t8", {})
                                s8 = util.do_sql(conn, "SELECT * FROM t8 WHERE
                                    id = :x", {"x": util.get_one_data(s7, 't7',
                                    'val')})
                                if s8:
                                    s9_ = util.do_sql(conn, "SELECT * FROM t9",
                                        {})
                                    s9 = util.do_sql(conn, "SELECT * FROM t9
                                        WHERE id = :x", {"x": util.get_one_data(
                                        s8, 't8', 'val')})
```

G.5 Ambiguous Short Reference Chains (AS)

Version 1:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
```

Version 2:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
```

Version 3:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
```

Version 4:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    sys.argv[4]})
                print(util.get_data(s4, 't4', 'val'))
```

Version 5:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    sys.argv[4]})
                print(util.get_data(s4, 't4', 'val'))
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": sys.argv[5]})
                    print(util.get_data(s5, 't5', 'val'))
```

Version 6:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    sys.argv[4]})
                print(util.get_data(s4, 't4', 'val'))
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": sys.argv[5]})
                    print(util.get_data(s5, 't5', 'val'))
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": sys.argv[6]})
                        print(util.get_data(s6, 't6', 'val'))
```

Version 7:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    sys.argv[4]})
                print(util.get_data(s4, 't4', 'val'))
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": sys.argv[5]})
                    print(util.get_data(s5, 't5', 'val'))
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": sys.argv[6]})
                        print(util.get_data(s6, 't6', 'val'))
                        if s6:
                            s7_ = util.do_sql(conn, "SELECT * FROM t7", {})
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": sys.argv[7]})
                            print(util.get_data(s7, 't7', 'val'))
```

Version 8:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    sys.argv[4]})
                print(util.get_data(s4, 't4', 'val'))
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": sys.argv[5]})
                    print(util.get_data(s5, 't5', 'val'))
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": sys.argv[6]})
                        print(util.get_data(s6, 't6', 'val'))
                        if s6:
                            s7_ = util.do_sql(conn, "SELECT * FROM t7", {})
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": sys.argv[7]})
                            print(util.get_data(s7, 't7', 'val'))
                            if s7:
                                s8_ = util.do_sql(conn, "SELECT * FROM t8", {})
                                s8 = util.do_sql(conn, "SELECT * FROM t8 WHERE
                                    id = :x", {"x": sys.argv[8]})
                                print(util.get_data(s8, 't8', 'val'))
```

Version 9:

```
import sys
import active_utils as util

with util.open_database('active_test_app') as conn:
    s1_ = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t1 WHERE id = :x", {"x": sys.argv[1]})
    print(util.get_data(s1, 't1', 'val'))
    if s1:
        s2_ = util.do_sql(conn, "SELECT * FROM t2", {})
        s2 = util.do_sql(conn, "SELECT * FROM t2 WHERE id = :x", {"x": sys.argv
            [2]})
        print(util.get_data(s2, 't2', 'val'))
        if s2:
            s3_ = util.do_sql(conn, "SELECT * FROM t3", {})
            s3 = util.do_sql(conn, "SELECT * FROM t3 WHERE id = :x", {"x": sys.
                argv[3]})
            print(util.get_data(s3, 't3', 'val'))
            if s3:
                s4_ = util.do_sql(conn, "SELECT * FROM t4", {})
                s4 = util.do_sql(conn, "SELECT * FROM t4 WHERE id = :x", {"x":
                    sys.argv[4]})
                print(util.get_data(s4, 't4', 'val'))
                if s4:
                    s5_ = util.do_sql(conn, "SELECT * FROM t5", {})
                    s5 = util.do_sql(conn, "SELECT * FROM t5 WHERE id = :x", {"x
                        ": sys.argv[5]})
                    print(util.get_data(s5, 't5', 'val'))
                    if s5:
                        s6_ = util.do_sql(conn, "SELECT * FROM t6", {})
                        s6 = util.do_sql(conn, "SELECT * FROM t6 WHERE id = :x",
                             {"x": sys.argv[6]})
                        print(util.get_data(s6, 't6', 'val'))
                        if s6:
                            s7_ = util.do_sql(conn, "SELECT * FROM t7", {})
                            s7 = util.do_sql(conn, "SELECT * FROM t7 WHERE id =
                                :x", {"x": sys.argv[7]})
                            print(util.get_data(s7, 't7', 'val'))
                            if s7:
                                s8_ = util.do_sql(conn, "SELECT * FROM t8", {})
                                s8 = util.do_sql(conn, "SELECT * FROM t8 WHERE
                                    id = :x", {"x": sys.argv[8]})
                                print(util.get_data(s8, 't8', 'val'))
                                if s8:
                                    s9_ = util.do_sql(conn, "SELECT * FROM t9",
                                        {})
                                    s9 = util.do_sql(conn, "SELECT * FROM t9
                                        WHERE id = :x", {"x": sys.argv[9]})
                                    print(util.get_data(s9, 't9', 'val'))
```

ACKNOWLEDGMENTS

We thank Jürgen Cito, Shivam Handa, Osbert Bastani, Cong Yan, Sara Achour, Deokhwan Kim, James Koppel, Rahul Sharma, and the anonymous reviewers for their insightful and helpful comments.

REFERENCES

[1] 2018. Enki. Retrieved from https://github.com/xaviershay/enki.

[2] 2018. Fulcrum. Retrieved from https://github.com/fulcrum-agile/fulcrum.

[3] 2018. Getting Started with Rails. Retrieved from http://guides.rubyonrails.org/getting_started.html.

[4] 2018. Kandan – Modern Open Source Chat. Retrieved from https://github.com/kandanapp/kandan.

[5] 2019. PLDI 2019 Konure Code. Retrieved from http://people.csail.mit.edu/jiasi/pldi2019.code/.

[6] 2020. Software Assurance Reference Dataset. Retrieved from https://samate.nist.gov/SARD/testsuite.php.

[7] F. Aarts, J. De Ruiter, and E. Poll. 2013. Formal models of bank cards for free. In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*. 461–468. DOI: https://doi.org/10.1109/ICSTW.2013.60

[8] Fides Aarts and Frits Vaandrager. 2010. *Learning I/O Automata*. Springer Berlin, 71–85. DOI: https://doi.org/10.1007/978-3-642-15375-4_6

[9] M. H. Alalfi, J. R. Cordy, and T. R. Dean. 2009. WAFA: Fine-grained dynamic analysis of web applications. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution*. 141–150. DOI: https://doi.org/10.1109/WSE.2009.5631226

[10] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD'13)*. 1–8.

[11] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Inf. Comput.* 75, 2 (Nov. 1987), 87–106. DOI: https://doi.org/10.1016/0890-5401(87)90052-6

[12] Dana Angluin and Carl H. Smith. 1983. Inductive inference: Theory and methods. *ACM Comput. Surv.* 15, 3 (Sept. 1983), 237–269. DOI: https://doi.org/10.1145/356914.356918

[13] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2007. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the ACM Conference on Computer and Communications (CCS'07)*. 13. DOI: https://doi.org/10.1145/1315245.1315249

[14] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 123–134. DOI: https://doi.org/10.1145/2442516.2442529

[15] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. 95–110.

[16] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 678–692. DOI: https://doi.org/10.1145/3192366.3192383

[17] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2015. Recursive games for compositional program synthesis. In *Proceedings of the 7th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'15)*. 19–39.

[18] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2010. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.* 13, 2 (Mar. 2010). DOI: https://doi.org/10.1145/1698750.1698754

[19] James F. Bowring, James M. Rehg, and Mary Jean Harrold. 2004. Active learning for automatic classification of software behavior. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. ACM, New York, NY, 195–205. DOI: https://doi.org/10.1145/1007512.1007539

[20] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Science & Business Media.

[21] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*. ACM, New York, NY, 322–335. DOI: https://doi.org/10.1145/1180405.1180445

[22] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. 2019. Active learning for software engineering. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'19)*. Association for Computing Machinery, New York, NY, 62–78. DOI: https://doi.org/10.1145/3359591.3359732

[23] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Form. Asp. Comput.* 28, 2 (2016), 233–263. DOI: https://doi.org/10.1007/s00165-016-0355-5

[24] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 3–14. DOI: https://doi.org/10.1145/2491956.2462180

[25] T. S. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.* 4, 3 (May 1978), 178–187. DOI: https://doi.org/10.1109/TSE.1978.231496

[26] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An automated prover for SQL. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR'17)*. Retrieved from http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf.

[27] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. 2013. Dynamic program analysis for database reverse engineering. In *Generative and Transformational Techniques in Software Engineering IV*. Springer, 297–321.

[28] Yossi Cohen and Yishai A. Feldman. 2003. Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM Trans. Softw. Eng. Methodol.* 12, 3 (July 2003), 285–316. DOI: https://doi.org/10.1145/958961.958962

[29] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng.* 35, 5 (Sept. 2009), 684–702. DOI: https://doi.org/10.1109/TSE.2009.28

[30] Kathi Hogshead Davis and Peter H. Aiken. 2000. Data reverse engineering: A historical survey. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*. IEEE Computer Society, Washington, DC, 70–.

[31] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of TLS implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, 193–206.

[32] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2016. Sampling for Bayesian program learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1289–1297.

[33] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. 420–435.

[34] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. 422–436.

[35] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 229–239.

[36] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. *Combining Model Learning and Model Checking to Analyze TCP Implementations*. Springer International Publishing, Cham, 454–471. DOI: https://doi.org/10.1007/978-3-319-41540-6_25

[37] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. 2007. A static analysis framework for detecting SQL injection vulnerabilities. In *Proceedings of the IEEE Computer Society Computers, Software, and Applications Conference (COMPSAC'07)*. DOI: https://doi.org/10.1109/COMPSAC.2007.43

[38] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Washington, DC, 474–484. DOI: https://doi.org/10.1109/ICSE.2009.5070546

[39] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning commutativity specifications. In *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV'15)*. 307–323.

[40] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 213–223. DOI: https://doi.org/10.1145/1065010.1065036

[41] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. *Queue* 10, 1 (Jan. 2012). DOI: https://doi.org/10.1145/2090147.2094081

[42] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. 2010. Learning of event-recording automata. *Theor. Comput. Sci.* 411, 47 (Oct. 2010), 4029–4054. DOI: https://doi.org/10.1016/j.tcs.2010.07.008

[43] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Found. Trends Program. Lang.* 4, 1–2 (2017), 1–119.

[44] B. P. Gupta, D. Vira, and S. Sudarshan. 2010. X-data: Generating test data for killing SQL mutants. In *Proceedings of the IEEE 26th International Conference on Data Engineering (ICDE'10)*. 876–879. DOI: https://doi.org/10.1109/ICDE.2010.5447862

[45] Raju Halder and Agostino Cortesi. 2010. Obfuscation-based analysis of SQL injection attacks. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC'10)*. 931–938. DOI: https://doi.org/10.1109/ISCC.2010.5546750

[46] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. 174–183. DOI: https://doi.org/10.1145/1101908.1101935

[47] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. 710–720.

[48] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. *The TTT Algorithm: A Redundancy-free Approach to Active Automata Learning*. Springer International Publishing, Cham, 307–322. DOI: https://doi.org/10.1007/978-3-319-11164-3_26

[49] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 156–167.

[50] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: Sketching for Java. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE15)*. 934–937.

[51] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. ACM, New York, NY, 215–224. DOI: https://doi.org/10.1145/1806799.1806833

[52] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'06)*. 6. DOI: https://doi.org/10.1109/SP.2006.29

[53] V. Benjamin Livshits and Monica S. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the USENIX Security Symposium (SSYM'05)*. 18–18.

[54] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. 2012. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, Piscataway, NJ, 80–90. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337233.

[55] Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. 2012. Active refinement of clone anomaly reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, 397–407.

[56] Edward F. Moore. 1956. Gedanken—Experiments on sequential machines. *Autom. Stud.* 34 (1956), 129–153.

[57] Nesrine Noughi, Marco Mori, Loup Meurice, and Anthony Cleve. 2014. Understanding the database manipulation behavior of programs. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. ACM, New York, NY, 64–67. DOI: https://doi.org/10.1145/2597008.2597790

[58] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 408–418. DOI: https://doi.org/10.1145/2594291.2594297

[59] Jeff Perkins, Jordan Eikenberry, Alessandro Coglio, Daniel Willenson, Stelios Sidiroglou-Douskos, and Martin Rinard. 2016. AutoRand: Automatic keyword randomization to prevent injection attacks. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*. Springer-Verlag New York, Inc., New York, NY, 37–57. DOI: https://doi.org/10.1007/978-3-319-40667-1_3

[60] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. 2017. Assertion generation through active learning. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C'17)*. IEEE Press, Piscataway, NJ, 155–157. DOI: https://doi.org/10.1109/ICSE-C.2017.87

[61] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'16)*. 522–538.

[62] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Kaelbling. 2018. Selecting representative examples for program synthesis. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 4161–4170. Retrieved from http://proceedings.mlr.press/v80/pu18b.html.

[63] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černý. 2018. DroidStar: Callback typestates for Android classes. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 1160–1170. DOI: https://doi.org/10.1145/3180155.3180232

[64] Harald Raffelt, Bernhard Steffen, and Therese Berg. 2005. LearnLib: A library for automata learning and experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'05)*. ACM, New York, NY, 62–71. DOI: https://doi.org/10.1145/1081180.1081189

[65] George Reese. 2000. *Database Programming with JDBC and JAVA*. O'Reilly Media, Inc.

[66] Martin C. Rinard. 2007. Living in the comfort zone. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 611–622.

[67] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. 2018. Active learning for inference and regeneration of computer programs that store and retrieve data. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'18)*. ACM, New York, NY, 12–28. DOI: https://doi.org/10.1145/3276954.3276959

[68] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*. ACM, New York, NY, 263–272. DOI: https://doi.org/10.1145/1081706.1081750

[69] Burr Settles. 2009. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

[70] Jiasi Shen and Martin Rinard. 2018. *Using Dynamic Monitoring to Synthesize Models of Applications That Access Databases*. Technical Report. Retrieved from http://hdl.handle.net/1721.1/118184.

[71] Jiasi Shen and Martin Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM. DOI: https://doi.org/10.1145/3314221.3314591

[72] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of datalog programs. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. Association for Computing Machinery, New York, NY, 515–527. DOI: https://doi.org/10.1145/3236024.3236034

[73] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. 2015. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 473–486. DOI: https://doi.org/10.1145/2694344.2694389

[74] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. ACM, New York, NY, 404–415. DOI: https://doi.org/10.1145/1168857.1168907

[75] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. 2015. TesMa and CATG: Automated test generation tools for models of enterprise applications. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. 717–720. DOI: https://doi.org/10.1109/ICSE.2015.231

[76] Frits Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95. DOI: https://doi.org/10.1145/2967606

[77] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic query exploration. In *Formal Methods and Software Engineering*, Karin Breitman and Ana Cavalcanti (Eds.). Springer Berlin, 49–68.

[78] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin, 425–446.

[79] Michele Volpato and Jan Tretmans. 2015. Approximate active learning of nondeterministic input output transition systems. *Electron. Commun. EASST* 72 (2015).

[80] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, New York, NY, 452–466. DOI: https://doi.org/10.1145/3062341.3062365

[81] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2018. Speeding up symbolic reasoning for relational queries. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). DOI: https://doi.org/10.1145/3276527

[82] Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. Association for Computing Machinery, New York, NY, 677–682. DOI: https://doi.org/10.1145/2642937.2642947

[83] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30.

[84] Michael Widenius and Davis Axmark. 2002. *Mysql Reference Manual* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA.

[85] Jerry Wu. 2018. *Using Dynamic Analysis to Infer Python Programs and Convert Them into Database Programs*. Master's thesis. Massachusetts Institute of Technology, Cambridge, MA.

[86] Wenfei Wu, Ying Zhang, and Sujata Banerjee. 2016. Automatic synthesis of NF models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets'16)*. ACM, New York, NY, 29–35. DOI: https://doi.org/10.1145/3005745.3005754

[87] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. 508–521.

[88] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. DOI: https://doi.org/10.1145/3187009.3177735

[89] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query synthesis from natural language. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017). DOI: https://doi.org/10.1145/3133887
[90] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the ACM on Conference on Information and Knowledge Management (CIKM'17)*. ACM, New York, NY, 1299–1308.
[91] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. 2004. DiscoTect: A system for discovering architectures from running systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society, Washington, DC, 470–479.

Received May 2019; revised September 2020; accepted October 2020