Program Inference and Regeneration via Active Learning

by

Jiasi Shen

B.S., Peking University (2013)
S.M., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 26, 2022

Certified by . . . . . .
Martin C. Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Program Inference and Regeneration via Active Learning

by

Jiasi Shen

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Software now plays a central role in numerous aspects of human society. Current software development practices involve significant developer effort in all phases of the software life cycle, including the development of new software, detection and elimination of defects and security vulnerabilities in existing software, maintenance of legacy software, and integration of existing software into more contexts, with the quality of the resulting software still leaving much to be desired. The goal of my research is to improve software quality and reduce costs by automating tasks that currently require substantial manual engineering effort.

I present a novel approach for program inference and regeneration, which takes an existing program, learns its core functionality as a black box, builds a model that captures this functionality, and uses the model to generate a new program. The new program delivers the same core functionality but is potentially augmented or transformed to eliminate defects, systematically introduce safety or security checks, or operate successfully in different environments.

This research enables the rejuvenation and retargeting of existing software and provides a powerful way for developers to express program functionality that adapts flexibly to a variety of contexts. For instance, one benefit is enabling new development methodologies that work with simple prototype implementations as specifications, then use regeneration to automatically obtain clean, efficient, and secure implementations. Another benefit is automatically improving program comprehension and producing cleaner code, making the code more transparent and the developers more productive. A third benefit is automatically extracting the human knowledge crystallized and encapsulated in legacy software systems and retargeting it to new languages and platforms, including languages and platforms that provide more powerful features.

In this thesis, I present two systems that implement this approach for database-backed programs.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor, Martin Rinard. Being Martin's student has been a unique experience. He almost destroyed my research style and then rebuilt it. By listening to Martin's daily conversations, I realized that it might actually be fun to make bold claims and convince others to agree. I came to realize that the great fun of research was not in becoming incrementally better than prior work, but in being the first to think of and prove a brand new idea. I wanted to do this kind of research but did not know how to do it at the beginning of my Ph.D. Martin has taught me how to do it. He has guided me to identify important unsolved problems in our field. He has trained me to approach difficult problems by turning them into simpler variants. He has demonstrated his vision and boldness by waiting years until the initially unconventional ideas were gradually adopted by our community. Martin has inspired us to pursue high-risk, high-reward research directions, providing us with technical, financial, and social support to help us tolerate the risks. I am grateful to Martin for providing such a precious research environment.

I would like to thank my thesis committee, Saman Amarasinghe and Armando Solar-Lezama. Their thought-provoking comments have helped me think about how my research fits in a greater context. I would also like to thank them and Jürgen Cito, one of my collaborators, for helping me with my job applications. These professors as well as Adam Chlipala, who served on my research qualification exam committee, and Daniel Jackson, who served as my graduate counselor, challenged me with interesting philosophical questions that prepared me for a broader audience.

I would like to thank my labmates and collaborators, including Sara Achour, José Cambronero, Michael Carbin, Jürgen Cito, Thurston Dang, Austin Gadient, Michael Gordon, Shivam Handa, Malavika Samak, Kai Jia, Charles Jin, Fereshte Khani, Deokhwan Kim, Fan Long, Varun Mangalick, Sasa Misailovic, Christopher Musco, Jeff Perkins, Zichao Qi, Julia Rubin, Stelios Sidiroglou-Douskos, Phillip Stanley-Marbell, Felix Stutz, Nikos Vasilakis, Jerry Wu, Yichen Yang, Adam Yedidia, and Damien Zufferey. They are creative and open-minded and have helped me build my research

tastes, technical insights, and social skills. My thesis won't be possible without them.

Other members of the MIT PL/SE community have also inspired me in a variety of ways. Ad-hoc hallway chats were sometimes more productive than official meetings. I will miss the good times when I could simply run into these brilliant people on the 7th floor of Stata.

I would like to thank my fellow Ph.D. students, especially Cheng Peng and Tsui-Wei (Lily) Weng, for sharing happiness and sorrow when we were trying to figure things out along our journey.

Special thanks to my parents, my grandparents, and Yibo Gao for their support. Their curiosity and perseverance have given me strength and inspiration over the years.

Contents

List of Figures

List of Tables

Chapter 1

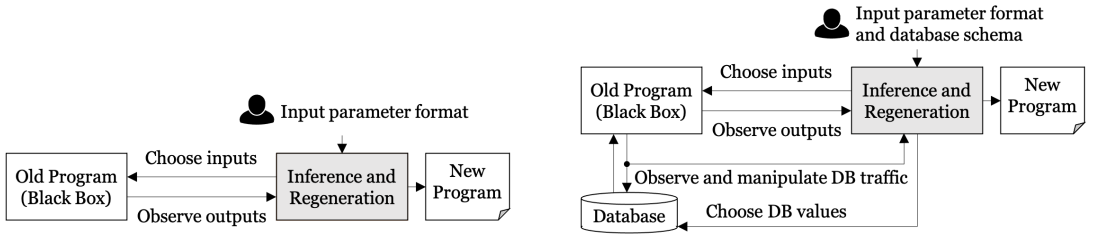
Introduction

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.

This thesis presents techniques for analyzing software to extract its core functionality and formulate this core functionality as a formal model. The goal is to make this model flexible, comprehensible, and exact. To achieve this goal, I identify recurring patterns in software, capture these patterns formally, and then exploit properties of these patterns to develop targeted *program inference algorithms* that use active learning to infer the model of the target program.

This approach involves the following key components:

Domain of Computation. The approach involves a domain of supported programs whose behavior is defined formally via a domain-specific language (DSL). The DSL characterizes the externally observable behavior of programs that can be inferred. Generally, the externally observable behavior consists of the input-output behavior of the program (Figure 1-1a).



(a) Generic concept (b) Specialized for database-backed programs

Figure 1-1: Program inference and regeneration via active learning

This thesis presents techniques for inferring programs that interact with an external database, whose the externally visible behavior also includes the database query traffic during execution (Figure 1-1b). 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 [85, 25, 34, 84, 114, 104, 95, 71]. Such applications therefore comprise a particularly compelling target for automatic program inference and regeneration.

Inference Algorithm. The approach involves an algorithm for inferring programs in the target domain. This algorithm exploits the domain knowledge to make the inference problem feasible. This thesis presents inference algorithms that perform *active learning* to eliminate uncertainty efficiently. In other words, the algorithms automatically select the inputs to run a program and infer its behavior. The inputs are selected to be the useful ones that distinguish nonequivalent programs in the target domain. These algorithms are efficient even when working with an infinite space of potential inputs and candidate programs, and they infer a uniquely correct model in the DSL that is observationally equivalent to the original program, as long

as the behavior of the original program is expressible in the DSL.

Correctness Guarantee. We provide correctness guarantees for programs whose behavior is expressible in our DSL. Our approach involves a rigorous mathematical proof that the inference algorithm will terminate with a uniquely correct model, which is the same program as the original one as expressed in the DSL. A key enabler of this property is that the DSL has a *canonical form*. That is, when two programs in the DSL are observationally equivalent, they must be syntactically identical except for equivalent variable references. This characteristic enables us to prove that the inference algorithms are sound and complete.

Augmented Regeneration. In general, we expect the extracted models to have many potential uses in the software development life cycle. In this thesis I focus on the *automatic regeneration* of new programs that implement the original core functionality, but are potentially augmented or transformed to operate successfully in new software environments or platforms, contain systematically inserted security checks, or consist of newly generated code that is clean, maintainable, and/or expressed in a modern target programming language.

This model can then enable the automatic regeneration of new programs that deliver the same core functionality but are potentially augmented or transformed to operate successfully in different environments, such as by retargeting to various platforms, inserting systematic security checks, or generating new code that is clean and maintainable. These capabilities can help developers automatically manipulate and transform software, thereby reducing manual engineering effort.

1.1 Konure

Chapters 3, 4, and 5 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 DSL captures database-backed programs with the following key characteristics: (1) each statement performs an SQL query to retrieve rows from the database, (2) the retrieved rows determine the control flow, and (3) the data flow is largely visible in the database traffic.

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 that users may write programs directly in the DSL.

Inference Algorithm. The Konure algorithm uses an SMT solver to generate useful inputs and database values with which to execute the original program. As the algorithm executes the program, it observes the program behavior in terms of the database traffic and the outputs. Based on this observation, the algorithm (conceptually) maintains a hypothesis of the inferred program structure as a sentential form [16] (a partially expanded syntax tree) of the grammar that defines the DSL.

The algorithm recursively expands nonterminal symbols in the current working sentential form. Each recursive step of the algorithm identifies a unique correct production for expanding a nonterminal symbol in the DSL. Specifically, the algorithm selects a nonterminal in the hypothesis, 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.

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.

Key Inferrability Properties. The design of the Konure DSL, together with its associated top-down inference algorithm, is a central contribution of this thesis. 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.

Contributions. The KONURE research 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.

- **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 commands are chosen from five open-source applications: Fulcrum Task Manager [3], Kandan

Chat Room [5], Enki Blogging Application [2], Blog [4], and a student registration application developed by an independent evaluation team to test SQL injection attack detection and nullification techniques. Our results highlight KONURE's ability to infer and regenerate robust, safe Python implementations of commands originally coded in other languages.

1.2 SHEAR

With KONURE, an important open question is how to effectively infer programs that contain loops and repetitive structures. A common approach in KONURE and other prior work is to apply heuristics that attempt to recognize repetitive structures in execution traces, then use those structures to infer the underlying presence of loops. Because there are, in general, multiple program structures that can generate the observed repetitions, prior techniques all limit either the program structures or the observed sequences that they can successfully work with. For example, the KONURE DSL allows only a few limited forms of multiple similar queries to occur in programs. It also requires any loop to be the last statement of the program. And it does not support nested loops. These limitations can constrain the applicability of prior techniques (Section 7.1.3).

Chapters 6 and 7 present a new system, SHEAR, that addresses the loop inference problem. SHEAR significantly expands the DSL to support more expressive loop and repetition structures than KONURE while preserving the correctness guarantees.

SHEAR uses a novel dynamic analysis technique based on *speculative manipulation*. Our technique works with database-backed programs, observing and manipulating the interactions with the database to infer the presence and structure of loops in the program. This capability enables SHEAR to expand the range of inferrable computations that may involve loop and repetition structures.

Inference Algorithm. The SHEAR algorithm is based on the following key ideas.

- **Probe-and-Validate Cycle:** Our technique first identifies program execu-

tion points that may correspond to loops in the program that iterate over rows returned from database queries. It then repeatedly executes the program on the same input, but systematically intervenes in the interactions between the program and the database by changing queries presented to the database to manipulate the number of rows that the database returns in response to the query. This intervention systematically elicits different program behaviors that expose information about the program structure, enhancing our ability to observe internal program structures and enabling us to efficiently infer the presence and structure of loops.

- **Fine-Grained Interactive Feedback:** The probe-and-validate cycle enables our technique to go beyond merely observing unaltered program executions. Actively manipulating the interactions with the environment enables the technique to purposefully generate new, altered, and otherwise unrealizable executions that expose additional information about the internal structure of the program. The result is a more powerful observation mechanism that enables our technique to formulate and efficiently resolve more expressive hypotheses about the internal structure of the program.

- **Modular Constructive Inference:** The fine-grained interactive feedback enables our technique to identify and fully disambiguate loop and repetitive structures in our target programs, many of which are otherwise indistinguishable given tractable numbers of unaltered program executions. The enhanced observational power also enables an efficient, top-down inference algorithm that unambiguously resolves each successive program construct in turn with no need to backtrack or explore a larger program search space. The result is an algorithm that linearly infers the full program by performing one inference step for each program structure.

We characterize the range of programs that work with Shear using a domain-specific language (DSL). For any program that is expressible in this DSL, Shear un-

ambiguously infers correct program structures—including loops and repetitive structures—that are identical to the ground truth.

We evaluate our technique based on its ability to infer and regenerate database-backed programs. The evaluation indicates that SHEAR is able to infer and regenerate a much wider range of programs than prior work [110, 135].[1] Our evaluation also indicates that our algorithm is scalable and efficient, which highlights the broader applicability of our dynamic analysis approach.

Contributions. The SHEAR research makes the following contributions.

- **Speculative Manipulation:** It presents a dynamic analysis approach based on speculative manipulation, which alters a program's interaction with the environment to infer the program structure. To the best of our knowledge, this is the first technique that analyzes a program by intervening in the program executions. This novel approach enables SHEAR to address loop-related ambiguities, an open question in program inference.

- **Soundness and Completeness:** It presents a theorem that states that if the behavior of a program conforms to the DSL, then SHEAR infers the correct program. SHEAR infers the program by exploiting its strong boundary for interacting with the external database. SHEAR works well with programs written in any language or implementation styles, as long as the externally observable behavior of the programs can be expressed in the SHEAR DSL.

- **Experimental Results:** It presents experimental results from our SHEAR implementation. We evaluate SHEAR's expressiveness by using it to infer and regenerate open-source applications in Java and Ruby on Rails. SHEAR successfully infers computations that go beyond the scope of prior work. We also evaluate the scalability and efficiency of our algorithm. These results highlight the effectiveness of our approach in inferring loop and repetitive structures.

[1] The most closely related prior work is Konure [135], which also infers programs that interact with relational databases, but only by observing unaltered program executions. SHEAR infers a strict superset of the programs that Konure infers. See Chapter 7 for a detailed analysis of the differences between Konure and SHEAR.

1.3 Design Rationale

Our approach is founded on several principles:

Programs as Specifications. We propose to use programs as (partial, noisy) specifications of the desired core functionality. In comparison with using standard specification languages based on formal logic, programming is a relatively widely available skill within our society (and a skill that promises to become more available over time). In comparison with natural language specifications, programs provide precision and the ability to explore and learn the specification by observing the program as it produces outputs in response to targeted synthesized inputs.

Noisy, Partial Programs. Developing a fully correct program that handles every corner case correctly is known to be much more difficult than developing a program that implements most of the desired functionality correctly. The inference algorithms are therefore designed to work with such mostly correct programs, in some cases by working with inputs that are unlikely to trigger rare corner cases, in others by identifying and discarding undesirable behavior (noise) from the original program that should not be part of the specification. Thus, this approach can work with original programs that implement the desired core functionaliry, potentially omitting error-checking and corner-case code, which can be inserted automatically during regeneration.

Focused Domains. To make the inference tractable, each inference algorithm focuses on a specific domain. In this thesis, we work with domains that are *finitely testable*, i.e., each computation in the domain can be uniquely identified from within all of the computations in the domain by a finite set of inputs. In this thesis we capture the domain via a domain-specific language.

Reinterpretation. Many modern programming languages support a simple and basic model of computation (sequential execution, file input and output, standard data structures, a single address space) that usually enables straightforward imple-

mentation of the desired core functionality. In many cases, however, the goal is to implement this core functionality in a more complex environment — to operate on distributed data, to work with data stored in a relational database or key/value store, to access specialized computing devices, to execute time-consuming computations in parallel, to package the core functionality into an appealing graphical user interface potentially accessed via the Internet, or to access values available via remote sensors. To support such implementations, the regeneration algorithm reinterprets standard constructs to translate them into implementations that operate successfully in the new, more complex target context.

Encapsulated Knowledge. The regenerator encapsulates the knowledge of how to use the complex software components that the regenerated code uses. Over the last several decades, the field has explored a variety of approaches for capturing and communicating this kind of knowledge. Examples include user manuals, textbooks, example programs, and, more recently, web sites such as Stack Overflow [1]. All of these mechanisms require the developer to examine the provided code and modify it to adapt it for their purpose in their system. Regeneration enables the developer to immediately obtain working code that implements the desired core functionality without the need to examine and/or modify the code (although the developer may very well do so if he or she desires). In this sense the regenerator can provide a more robust encapsulation of the (in some cases quite involved) knowledge required to productively use the powerful but complex target components.

1.3.1 Observer Models

A key trade-off point in the design of inference algorithms is the strength of the observer that collects the program behavior. Programs that are observationally equivalent when the observer is weaker may become distinguishable by a stronger observer. A stronger observer therefore enables an inference algorithm that is more precise and efficient but may also produce low-level detail that is less relevant for the target use case.

Monolithic Black Box. We previously developed a black-box inference algorithm that interacts with an original program by generating inputs and observing the resulting outputs [126]. An advantage of treating the program as a black box is that the original program can use any language or implementation methodology. For example, the black-box approach readily works with obfuscated programs.

Gray Box Consisting of Black-Box Components whose Interactions are Observable. Konure [135, 138] takes a gray-box approach. Konure treats a database-backed application as a gray box that consists of two main components that interact with each other. One such component is the database. The other component is the controller that implements the core business logic. This separation of components is appropriate for our target domain of database-backed applications, since these applications in practice are often structured as code bases that implement the business logic and interact with a third-party database server such as MySQL [165]. Konure infers the controller component as a black box by observing its inputs, outputs, and interactions with the database. By treating the controller component as a black box, Konure works directly with applications that would be difficult to analyze otherwise, such as ones that are obfuscated, built with complicated frameworks, or written in multiple languages. Meanwhile, because the database traffic is visible to Konure, it distinguishes nonequivalent programs more efficiently than when only the end-to-end inputs and outputs are observable.

Gray Box Consisting of Black-Box Components whose Interactions can be Overwritten. Shear [136] also takes a gray-box approach, but its observer is stronger than that of Konure. In addition to observing the inputs, outputs, and database interactions as Konure does, Shear also intervenes in the database interactions while the program executes. Shear uses such intervention to obtain fine-grained interactive feedback, which enables a more precise and efficient algorithm for inferring loop and repetition structures.

White Box. White-box approaches can also be appropriate, but may require more involved mechanisms that (dynamically or statically) instrument and/or analyze aspects of the program such as the source code or the binaries.

1.4 Benefits

Our approach enables a novel paradigm, *automatic software rejuvenation*, which takes an existing program, learns its core functionality, formulates it as a precise model, and uses the model to generate a new program. It enables the rejuvenation and retargeting of existing software. This paradigm can deliver benefits in many aspects of the software life cycle.

New Software. The first benefit is enabling new development methodologies that work with simple prototype implementations as (potentially noisy) specifications, then use regeneration to automatically obtain clean, efficient, and secure implementations specialized for the specific context into which they will be deployed.

One example is starting from simple prototypes that implement only the core functionality and generating correct-by-construction code augmented with checks that eliminate security vulnerabilities [126]. In this thesis, I present techniques that regenerate data-retrieval programs to add systematic checks that prevent SQL injection vulnerabilities [135, 138, 136]. My collaborators and I have also used the DSL-based approach to regenerate string-processing libraries, where the new versions no longer invoke original potentially dangerous dependencies, eliminating software supply-chain vulnerabilities [154].

Our approach promises to substantially reduce the time and effort required to obtain programs that work with complex programming interfaces on modern complex hardware platforms. By automating the generation of error, privacy, and security checking code, it promises to improve program robustness and reliability.

Program Comprehension. The second benefit is automatically improving program comprehension and producing cleaner code, making the code more transparent

and the developers more productive. Our techniques produce models of a program that characterizes the program's externally visible behavior, rather than its internal implementation details [135, 138, 136, 154, 126, 137]. Because the model captures core application functionality, it can help developers explore and better understand this functionality especially when the source code is obscure, obfuscated, or unavailable. Preliminary results indicate that these models can help developers understand the business logic flowing across multiple source files built with sophisticated frameworks [55].

More broadly, our approach brings benefits to software archaeology, where the developer starts with a legacy system that implements the desired functionality. Here one use case is to start with a system that runs in an obsolete or otherwise undesirable computing context to obtain a regenerated version that can operate successfully in a more modern context. Another use case is to start with a system that may have defects or security vulnerabilities to generate a program without defects or vulnerabilities (by, for example, systematically generating appropriate checks and code). Yet another use case is to improve performance by replacing an inefficient implementation with a more efficient regenerated implementation. Even another use case is to replace a program that has been maintained so intensively that its lifecycle is over and it is no longer feasible to continue to maintain it [30].

Targeted Functionality Extraction. The third benefit is automatically extracting the human knowledge in software and retargeting it to different languages and platforms that provide more powerful features. Conceptually, the original program implements a range of functionality, only part of which comprises the desired core functionality. The developer provides a limited interface specification that targets only the desired core functionality. The remaining undesired functionality is discarded by the program inference and regeneration system, which then generates additional functionality in the new version of the program.

For example, our techniques regenerate data storage and retrieval programs, where the new versions are augmented with back-end databases and front-end web inter-

faces [126]. As another example, I supervised a Master's thesis that regenerates Python programs to use an external database instead of the original in-memory data structures [45, 167]. Our technique can also regenerate stream-processing programs with parallelism, so that the new versions perform the same computations but run faster [137]. More broadly, our approach promises to enable developers to express software functionality in flexible new ways that adapt to a variety of contexts.

Chapter 2

Related Work

In this chapter, we discuss related work in active learning, program synthesis, reverse engineering, and program transformation techniques such as code translation, refactoring, and optimization.

2.1 Active Learning

Machine Learning. Active learning is a classical topic in machine learning [133]. Machine learning algorithms learn a relationship between features and predictions by extracting statistical information from training data. Our approach similarly generates programs that are consistent with the given implementation, and we anticipate the possibility of applying techniques related to program synthesis or machine learning. In contrast to most approaches in the machine learning community [133, 124], our approach symbolically reasons about potential program structures and obtains unique correct answers. 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.

State Machine Model Learning. State machine learning algorithms [108, 23, 53, 120, 78, 13, 89, 157, 48, 70, 153, 12, 62, 119, 169] construct partial representations

of program functionality in the form of finite automata with states and transition rules. State fuzzing tools [12, 62, 119] hypothesize state machines for programs. Network function state model extraction [169] uses program slicing and models the sliced partial programs as packet-processing automata. These algorithms extract partial models of the given programs. Our approach, in contrast, (1) extracts a complete representation of the core functionality, which, in turn, enables the regeneration (and replacement) of the initial program, (2) can work with programs with defects or that only partially implement the core functionality, (3) regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms, and (4) represents the inferred programs in a DSL, which can capture a wide range of programs that access databases.

Stateless Model Extraction. Model extraction algorithms use queries to construct representations for programs, where the representations are stateless functions such as decision trees [151, 59] or symbolic rules [150]. Model compression algorithms [39, 88] use machine learning models, such as neural networks, to mimic a machine learning model, often by generating inputs (training data) and observing the outputs from the given model. Our approach, in contrast, (1) infers stateful models that retrieve data across multiple queries and (2) regenerates a new program or programs, augmented as appropriate, that implement the core functionality on new implementation platforms.

Learning the Core Program Functionality. Our previous research produced active learning techniques for black-box inference of programs that manipulate key/-value maps [126] and string-processing libraries [154]. Konure and Shear, in contrast, also observe database traffic, work with broader and more expressive classes of applications, and deploy top-down, syntax-guided inference algorithms (as opposed to enumerating store/retrieve pairs as in [126]). 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 [45, 167]. Konure and Shear, in contrast, observe the use of an existing external database, work with programs implemented in any programming language, and guarantee sound and complete inference for programs in the Konure and Shear DSLs (as opposed to providing probabilistic correctness properties as in [45, 167]). Shear also differs from all these techniques in that it manipulates the program execution in addition to observing it.

Oracle-guided synthesis as implemented in Brahma [92] interacts with a program to infer a model that completely captures the behavior of the program. Brahma implements oracle-guided synthesis for loop-free programs that compute functions of finite-precision bit-vector inputs. Brahma finitizes the synthesis problem by working with a finite set of components, with each component used exactly once in the synthesized model. Our approach, in contrast, works with an unbounded space of programs that may contain nested control flow and loops that iterate over the results of database queries. Brahma adopts a flat, solver-based approach that repeatedly 1) generates two programs that both satisfy the current set of input/output pairs, 2) generates a new input that distinguishes the two programs, 3) queries an oracle to find the correct output for the new input, and 4) adds the resulting input/output pair to the current set of input/output pairs. Brahma terminates when there is only one program that satisfies the set of input/output pairs. Our approach, in contrast, maintains a structured representation that captures the inferred program structure. Our approach deploys an inference algorithm guided by assumptions about the inferrable program behavior defined as a DSL. In particular, Konure and Shear maintain a sentential form that captures all remaining possible models and refines the hypothesis by generating inputs and database configurations to determine which production to apply to the current nonterminal in the sentential form. This structured approach is very efficient when narrowing down an unbounded space of programs.

Mimic [87] traces the memory accesses of an opaque function to synthesize a model of the traced function. It uses a random generate-and-test search over a space

of programs generated by code mutation operators, with a carefully designed fitness function measuring the degree to which the current model matches the observed memory traces. Input generation heuristics are used to find inputs that work well with the mutation operators and fitness function to find suitable code models. There is no guarantee that the generated model is correct or that the search will find a model if one exists. Mimic was applied to infer models for the Java Arrays.prototype computations, successfully inferring models for 12 of these computations. In contrast, our approach targets a different class of computations, which enables it to deploy an algorithm that is guaranteed to infer a model if the application conforms to the required domain.

Learning Specific Aspects of Program Behavior. ALPS uses active learning to prune the search space for synthesizing Datalog programs, which consist of rules [139]. Konure and Shear, in contrast, work 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 [74], a technique for learning program input grammars [27], a technique for learning points-to specifications [28], a technique for learning models of the design patterns that Java computations implement [90], a technique for learning classifiers for event-transition behavior [36], and a technique for inferring the input parsing functionality of programs [45]. Unlike Konure and Shear, 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 [161], ranking anomaly reports [106], and improving candidate assertions [115].

Generating Inputs to Discover Defects. Concolic testing [132, 75, 44, 76] generates inputs that systematically explore all execution paths in the program. The goal is to find inputs that expose software defects. BuzzFuzz [73] generates inputs that

target defects that occur because of coding oversights at the boundary between application and library code. DIODE [140] generates inputs that target integer overflow errors. All of these techniques dynamically analyze the execution of the program and use the resulting information to guide the input generation. They all 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. Simian [33] synthesizes multi-client interactions to trigger defects such as conflicts in collaborative web applications. Our approach, in contrast, (1) works with the given implementation as a black box, without analyzing code, (2) works with applications whose behavior can be productively modeled with programs in our DSL, (3) infers a model that captures the complete core functionality of the program, and (4) regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms.

2.2 Program Synthesis

The vast majority of program synthesis research works with a given set of input/output examples [145, 68, 91, 170, 116, 69, 31, 20, 67, 164, 82, 65, 117, 171, 172, 158, 141, 80, 77, 35, 113]. 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 [91] or a heuristic that ranks synthesized programs (for example, ranking shorter programs above longer programs) [69, 82, 65]. Our approach, in contrast, uses active learning and speculative manipulation to reverse engineer an existing program (in effect using the original program as a specification), without requiring user interaction and is not restricted to a limited set of training data. Because this approach is not constrained by a given set of input/output pairs, it can select the inputs the inputs and database contents that purposefully target and resolve ambiguities, eliminate uncertainty, and obtain a model that completely captures the core application functionality. Another

key difference is that our approach is designed for generating better implementations after modeling the core functionality of the original implementation. Our approach therefore does not aim at strictly following all details of the specification (i.e., the original implementation). This design can be especially valuable for tolerating noise, such as corner cases overlooked by developers, in the original implementation.

Other synthesis techniques often involve solving constraints [94, 81], applying templates [145, 148, 147], or using hypothetical I/O oracles [92] to solve for programs that satisfy the specification. These techniques do not work with existing implementations, but require abstract specifications in other forms. Our approach, in contrast, (1) works with a concrete program implementation rather than abstract specifications, which can be especially useful for regenerating new implementations for legacy software, (2) actively and automatically executes the given implementation as needed to infer the core functionality, without requiring user interaction and is not restricted to a limited set of training data, (3) can work with programs with defects or that only partially implement the core functionality, and (4) automatically regenerates a new program or programs, augmented as appropriate, that implement the core functionality without defects or security vulnerabilities on new implementation platforms.

SyGus [20] identifies a range of program synthesis problems for which it is productive to structure the search space as a domain-specific language and presents a framework for this approach. Our approach similarly uses a domain to structure the search space. Unlike the examples presented in [20], our approach exploits the structure of the domain to obtain an inference algorithm that uses active learning to progressively refine the knowledge of the program structure. In particular, Konure and Shear deploy top-down inference algorithms that progressively refine 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 and Shear work effectively with an unbounded space of models.

LaSy works with a sequence of user-provided input/output pairs to iteratively generalize an overspecialized program [113]. 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.

Synthesizing Loops, Regular Expressions, and Recursions. In contrast to other systems that synthesize loops by observing executions of an existing program [32, 87, 118, 64], Shear (1) performs speculative manipulation to obtain fine-grained interactive feedback, (2) fully and precisely identifies loop structures in the program, and (3) constructs the loop structures modularly, without requiring enumerative search.

A fundamental difference between Shear and existing regular expression synthesis techniques is that Shear performs speculative manipulation to obtain an efficient modular constructive synthesis algorithm. There are two kinds of regular expression synthesis algorithms. Techniques that work only with positive and negative examples [102, 51, 175] provide no guarantees that they will infer the exact regular expression and rely on heuristics to deliver an ordered list of synthesized expressions. Shear, in contrast, produces a single correct loop structure within the Shear DSL. Techniques that do deliver a single correct regular expression require the ability to ask an oracle whether a candidate is correct [23]. Shear does not need this oracle, but instead works with an existing program.

Most other existing program synthesis techniques cannot explicitly synthesize the precise loop structures that produce a specified repetition behavior. These techniques work with loops in restricted ways. One approach used by these techniques is to use parameterized templates that contain known loop or recursive structures, followed by synthesizing loop-free expressions as parameters and synthesizing loop-free callers that invoke these templates as sealed building blocks [145, 100, 18, 92, 144, 146]. Another approach is to synthesize recursive or iterative functions—such as map, filter, fold, sum, and replace—that apply on data structures—such as lists and trees—with

known repetition boundaries [69, 142, 163, 14, 159, 80, 103, 50, 116, 111]. Yet another approach is to rank multiple nonequivalent loop candidates with heuristics [87, 51, 102, 143, 101, 112, 175, 162].

Synthesizing Models of Loops from High-Level Execution Traces. Inferring loops from repetitions has been a recurring problem in many areas, including program synthesis, program comprehension, performance profiling, and protocol reverse engineering. Prior systems that infer loops from program traces have used heuristics to partially address the loop inference problem. These systems either (1) do not attempt to fully identify the loop structure [87, 168, 41] or (2) impose restrictions to avoid dealing with ambiguous repetitions [110, 135, 99, 64]. Mimic [87] uses a probabilistic approach to rank candidate loops but does not guarantee identifying the correct loops. Nero [168] uses instrumentation to identify where each loop iteration starts in the trace, but does not identify where the last loop iteration ends unless it already knows the loop body. Dispatcher's [41, 42] "dynamic" loop detection technique detects repetitions from traces using heuristics and does not accurately determine where the loop ends in the trace. DaViS [110] uses heuristics to identify possible nested queries from the SQL trace, where the loop body contains exactly one SQL query. WebRobot [64] and Kobayashi's [99] algorithms are based on repetitions and would not work with programs that contain conditionals in a loop body or ambiguous repetitive instructions. In contrast, Shear infers the full loop structure unambiguously and removes many restrictions from prior work.

Memory Address Trace Compression. Many techniques identify potential loops in memory address traces, often to compress the traces for storage or to improve runtime performance of predicted loops [66, 40, 97, 127]. These techniques are often based on detecting linear progressions from the memory addresses in the traces. Shear, in contrast, does not require knowledge of the internal memory layouts.

Synthesizing Loops Based on Static Analysis or Low-Level Program State. There are many techniques that detect potential loops according to control flow

graphs, instruction addresses, memory addresses, stack frames, register values, taints, or other forms of low-level runtime information [152, 109, 43, 41, 42, 129, 86, 47, 98, 107, 14, 52, 96, 49, 26, 15]. In contrast to these techniques, SHEAR does not require analyzing the source code or low-level runtime information. Our approach, in contrast, observes only the SQL queries visible in the network traffic as the program communicates with an external database. As a result KONURE and SHEAR works well with applications written in any language or any implementation styles, as long as their externally observable behavior conforms to the corresponding DSL. Our approach also regenerates a more sophisticated version of a whole program with potentially added functionality.

2.3 Reverse Engineering

Techniques 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 [58, 22]. WAFA [17] 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 [110] visualizes the data-manipulation behavior of an execution of a data-intensive program. DAViS detects loops whose body contains only one query. DiscoTect [174] 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. Database reverse engineering analyzes a program's data access patterns, often to reconstruct implicit assumptions of the database schema [60, 56]. KONURE infers programs that interact with databases (and not the structure of the database).

2.4 Code Translation, Refactoring, and Optimization

Software Modernization. Software modernization [72, 46, 128] analyzes the source code of a legacy program, translates the program into a high-level modeling language, then uses this representation to generate a new program that implements the functionality in a more modern language. The translation strictly follows syntactic cues and usually requires human intervention. Our approach, in contrast, (1) works with the given implementation without analyzing code and (2) regenerates an augmented computation with additional error and security checks that implements the core functionality with complex new software components that execute on modern target platforms.

Partial Program Rejuvenation and Reengineering. Helium uses dynamic instrumentation to extract the functionality of computational stencil kernels embedded within production binaries [107]. It then replaces the stencil kernel with a computation expressed in Halide [121]. The goal is to replace the legacy implementation with a version optimized for modern computational platforms. Several other techniques [52, 26, 96, 15] also rewrite fragments of programs that perform loops into equivalent loop implementations to optimize the performance. Program fracture and recombination [21] works with multiple applications to automatically find efficient, sophisticated, and/or robust implementations of subcomputations across applications, then transfers subcomputations across implementations to maximize efficiency or robustness. A goal is to automatically replace simple code that executes on a single machine with more complex code that operates on parallel or distributed computing

platforms. Our approach, in contrast, (1) models the full data-retrieval computation and regenerates executable code, augmented as appropriate, (2) can work with incomplete or buggy implementations of the original program, and (3) targets programs that access databases.

Database program reengineering often involves analyzing the source code to produce equivalent but more efficient implementations [57, 52, 22]. In contrast, our approach (1) does not require 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 only transforming the individual database queries).

Model-Driven Engineering. In model-driven engineering (MDE) [38, 131, 130], developers specify functionality in high-level models, often using domain-specific languages or formalisms such as Unified Modeling Language (UML), which are then used to generate low-level platform-dependent implementations. Use cases of MDE include migrating software across different platforms [130] and automating the code generation for CRUD (create/read/update/delete) applications [19, 122, 63]. Like the regenerators presented in this paper, MDE code generators encapsulate the knowledge of how to use specific computational platforms and enable the automatic generation of code for multiple platforms. In contrast to having developers work directly with high-level domain-specific models, our approach starts from an existing implementation, then infers the program functionality as a black box and regenerates a new implementation.

Program refinement [166] is a software development strategy that gradually expands parts of a high-level implementation with more detailed versions.

Chapter 3

KONURE: Active Learning for Inference and Regeneration of Applications that Access Databases

In this chapter, we first present an example that illustrates how KONURE works. We then present the design and implementation of KONURE, including an overview, the KONURE DSL, the core inference and regeneration algorithm, and other implementation details.

3.1 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 [165] via JDBC [123].

Command. The application implements the following command: "`liststudentcourses -s` s `-p` p", where the input parameter s denotes student ID and p denotes pass-

word. 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. Figure 3-1 presents the main implementation of this command, which invokes the `passwordCheck` method in Figure 3-2 to check the student's password against the database. This method passes the input `password` to the database directly by string concatenation, causing the SQL injection vulnerability.

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 3-3) to observe the resulting database traffic, which the proxy collects as the *concrete trace* of the execution (Figure 3-5a). 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 3-5b). 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.

KONURE DSL. Figure 3-6 presents the (abstract) grammar for the KONURE DSL. A program consists of a sequence of Query statements potentially terminated by an If

```
public Set<Course> getRegistrationsForStudent(final Config config)
   throws SQLException {
  final Student studentreg = getStudent(config.getStudentId());
  final Set<Course> courses = new HashSet<Course>();
  Connection conn = null;
  PreparedStatement stmt = null, stmt2 = null, stmt3 = null;
  ResultSet rs = null, rs2 = null, rs3 = null;
  try {
    if (!passwordCheck(studentreg, config.getPassword())) {
      System.err.println("Incorrect Password");
    } else {
      conn = new DataConnection().initialize();
      stmt = conn.prepareStatement("SELECT * FROM course c "
            + "JOIN registration r on "
            + "r.course_id = c.id WHERE r.student_id = ?");
      stmt.setString(1, studentreg.getId());
      rs = stmt.executeQuery();
      while (rs.next()) {
        final String id = rs.getString(1);
        final String name = rs.getString(2);
        final String limit = rs.getString(3);
        final String courseNumber = rs.getString(4);
        final String isOffered = rs.getString(5);
        final String tid = rs.getString(6);
        String fname = null;
        String lname = null;
        stmt3=conn.prepareStatement("Select firstname, lastname
           from teacher where id = ?");
        stmt3.setString(1, tid);
        rs3 = stmt3.executeQuery();
        while (rs3.next()) {
          fname = rs3.getString(1);
          lname = rs3.getString(2);
        }
        final Course course = new Course(id, name, limit,
           courseNumber);
        course.setOffered(isOffered);
        course.setTeacher(new Teacher(tid,fname,lname));
        stmt2 = conn.prepareStatement("SELECT count(*) FROM
           registration WHERE course_id = ?");
        stmt2.setString(1, id);
        rs2 = stmt2.executeQuery();
        while (rs2.next()) {
          final String totalEnrolled = rs2.getString(1);
          course.setTotalEnrolled(totalEnrolled);
        }
        courses.add(course);
  } } } finally { closeResultSet(rs); closeResultSet(rs2);
     closeStatement(stmt); closeStatement(stmt2); conn.close(); }
  return courses;
}
```

Figure 3-1: Example command in Java

```
private boolean passwordCheck(Student unregstudent, String password
   ) throws SQLException {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = new DataConnection().initialize();
    stmt = conn.prepareStatement("SELECT * FROM student WHERE id=?
       AND password='" + password +"'");
    stmt.setString(1, unregstudent.getId());
    rs = stmt.executeQuery();
    if (rs.next()) {
      return true;
    } else {
      return false;
    }
  } catch (SQLException e) {
    System.err.println("sql error");
  } finally {
    closeStatement(stmt);
    conn.close();
  }
  return false;
}
```

Figure 3-2: Example command in Java (continued)

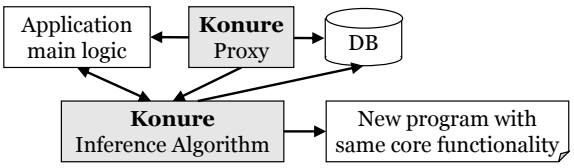


Figure 3-3: The KONURE architecture, including a transparent proxy interposed between the application and the database to observe the generated database traffic.

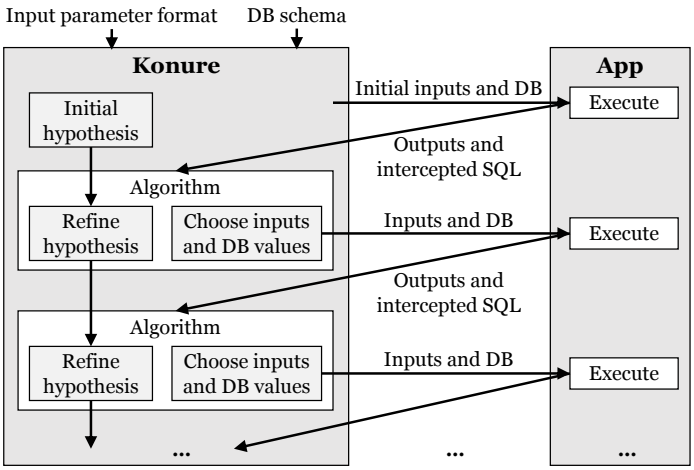


Figure 3-4: The KONURE active learning algorithm iteratively refines its hypothesis to infer the application.

```
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 3-5a. The conversion replaces the constant `'0'` with its origin location, the input parameter s.

Figure 3-5: First execution trace

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

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

Figure 3-6: Grammar for the KONURE DSL

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 3.3) are key to the inferrability of the DSL.

First Production. The first execution generated a single query (Figure 3-5a). KONURE determines if this query came from a Seq, If, or For statement as follows. Working with the abstract trace in Figure 3-5b, 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 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 3-5a and 3-7a present the recorded concrete traces; Figures 3-5b and 3-7b 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 (Fig-

```
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 3-7a. The conversion replaces the constants `'5'` and `'6'` with their origin locations, input parameters s and p.

Figure 3-7: Second execution trace

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

Figure 3-8: Hypothesis after resolving the topmost Prog nonterminal to an If statement.

ures 3-5a and 3-5b) or at least one row (Figures 3-7a and 3-7b). KONURE concludes that the first Query comes from an If statement and produces the first hypothesis in Figure 3-8. This hypothesis corresponds to applying an If production to the topmost Prog nonterminal.

Intuition. Recall that, in the KONURE DSL (Figure 3-6), there are four potential productions to apply for each Prog nonterminal: Prog $:= \epsilon$, 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 3-5 and 3-7). Applying the Seq production would result in a program that does not condition on the results of the first query (`q1` in Figures 3-5b and 3-7b), 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 3-5) but continues execution after the first query in the second

trace (Figure 3-7). 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 3-7b, 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 3-7a presents the trace from the execution in which the second query retrieves no rows; Figure 3-9a presents the trace from the execution in which the second query retrieves at least one row. Because the traces differ (similarly to the above First Production), KONURE resolves the nonterminal P_1 to an If statement. Figure 3-10 presents the resulting hypothesis.

Intuition. As with the topmost Prog nonterminal, the P_1 nonterminal (Figure 3-8) 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 3-8), which is incorrect because the program has produced traces that perform actions after the first query (`q1` in Figures 3-7b and 3-9b) retrieves nonempty data (Figures 3-7 and 3-9). 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 3-7b and 3-9b), 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 3-7) but continues execution after the second query in the third trace (Figure 3-9). Applying the For production to P_1 would result in a program that iterates over the rows retrieved by the first query in

```
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 3-9a. The conversion replaces the constants `'1'` and `'2'` with their origin locations, input parameters s and p.

Figure 3-9: 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
```

Figure 3-10: Hypothesis after resolving P_1 (Figure 3-8).

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 3-9b), 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 3-9b). 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 3-11.

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.

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 3-12) therefore contains a Print statement that prints these values.

Intuition. As with the previous Prog nonterminals, the P_3 nonterminal (Figure 3-10) also has four potential productions: Prog $\coloneqq \epsilon$, Prog $\coloneqq$ Seq, Prog $\coloneqq$ If, and Prog $\coloneqq$ For. For P_3, applying the ϵ production would result in a program with an empty inner `then` branch (Figure 3-10), which is incorrect because the program has produced traces that perform actions after the second query (`q2` in Figure 3-9b) retrieves nonempty data (Figures 3-9 and 3-11). 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 3-9b), 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 3-9) but continues execution after the third query in the fourth trace (Figure 3-11). 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 3-11). 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 detail in Section 3.3), so that 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

```
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'
```

Figure 3-11: Concrete trace from an execution to resolve P_3 (Figure 3-10). 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
```

Figure 3-12: Hypothesis after resolving P_3 (Figure 3-10).

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. Figure 3-13 presents the regenerated code. Here, each function call

```
util.do_sql(conn, query, params)
```

performs a database query. The query is constructed by (conceptually) replacing the variables in `query` with the corresponding values specified in `params` while systematically checking for errors in the SQL syntax. More concretely, the function `util.do_sql` invokes

```
conn.execute(sqlalchemy.text(query), params)
```

which uses the SQLAlchemy library to ensure that the values in `params` are passed to the database appropriately [29]. 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 [125, 105]. 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 outlines 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.

3.2 Design Overview

KONURE infers two aspects of the program. Firstly, starting from a concrete trace intercepted by the proxy (Figure 3-3), 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.

Secondly, starting from the unstructured sequences of queries in traces, KONURE infers the underlying control flow in the program. This inference algorithm is constructive [24] — 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 $\coloneqq \epsilon$, Prog $\coloneqq$ Seq, Prog $\coloneqq$ If, and Prog $\coloneqq$ For (Figure 3-6). 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.

3.3 Konure Domain-Specific Language

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

3.3.1 DSL Overview

We deisgn 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 3.3.4 and 3.3.5.

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.

3.3.2 Preliminaries

To formally define the KONURE DSL, we first define skeletons (Definition 3.3.1), then describe the externally observable part of a program (Definition 3.3.2), and finally define the DSL as a subset of programs in Prog (Figure 3-6) that satisfy two additional restrictions (Definition 3.3.3 and Definition 3.3.4). For readability, we present Definition 3.3.2 and Definition 3.3.3 here only at a high level and postpone their formalization to Chapter 4.

Definition 3.3.1. The *skeleton* of a program $P \in \text{Prog}$ is a program that is syntactically identical to P except for replacing syntactic components derived from the Orig nonterminal (Figure 3-6) with an empty placeholder $\diamond$. Figure 3-14 presents the syntax for skeleton programs. We write $\mathcal{S}$ for the set of skeleton programs, $\mathcal{S} = \text{SProg}$. For readability, we shall denote Prog elements (Figure 3-6) 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 in Figure 3-15. We use an overline, as in $\overline{C}$ and $\overline{O}$, to denote a list.

Clearly for any program $P \in \text{Prog}$, query $Q \in \text{Query}$, and expression $E \in \text{Expr}$, we have $\pi_{\mathcal{S}} P \in \mathcal{S}$, $\pi_{\mathcal{S}} Q \in \text{SQuery}$, and $\pi_{\mathcal{S}} E \in \text{SExpr}$.

Definition 3.3.2. 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 4.2.

Definition 3.3.3. 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.

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

$\mathcal{D}(P)$ 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 4.3.

3.3.3 DSL Definition

We present the grammar for the KONURE DSL in Figure 3-6. 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.

Definition 3.3.4 (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 3.4.2).[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 (Chapter 4).

We present several immediate extensions to $\mathcal{K}$ in Section 4.8.

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 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.

3.3.4 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

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

effectively observe the control flow execution by observing the database traffic (Figure 3-3). For example, the student registration program in Section 3.1 enters two different branches in the first two executions, which is inferrable by comparing the two corresponding traces (Figures 3-5 and 3-7).

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 that, 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 3.1) to enter a (potential) branch unvisited by the first execution (Figure 3-5), 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 3-7).

3.3.5 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 5.1.3. In general, KONURE is not designed to infer programs that cannot be captured by an inferrable DSL.

On the other hand, 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.

3.4 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 that the structure of the trace matches the corresponding path through the program.

3.4.1 Notation

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

Definition 3.4.1. 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 ($\sigma_R \in \mathit{Result}$):

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

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 3.4.2. In Section 3.1, the first execution of the program (Figure 3-5) uses the following context:

$$\sigma_1 = \langle \{s : \texttt{'0'}, p : \texttt{'1'}\}, \{\texttt{student} : \emptyset, \texttt{teacher} : \emptyset, \texttt{course} : \emptyset, \texttt{registration} : \emptyset\}, \emptyset \rangle.$$

This context sets input parameters s and p to 0 and 1, respectively, and sets all database tables to empty. The second execution (Figure 3-7) uses the following con-

text:

$$\sigma_2 = \langle \{s : \texttt{'5'}, p : \texttt{'6'}\},$$
$$\{\texttt{student} : \{1 : \{\texttt{id} : \texttt{'5'}, \texttt{password} : \texttt{'2'}, \texttt{firstname} : \texttt{'3'}, \texttt{lastname} : \texttt{'4'}\}\},$$
$$\texttt{teacher} : \emptyset, \texttt{course} : \emptyset, \texttt{registration} : \emptyset\}, \emptyset\rangle.$$

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

Definition 3.4.4. For an origin location $O \in$ Orig and a context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in$ *Context*, $\sigma(O)$ denotes the result from looking up O in σ. Specifically, for an input parameter $x \in$ *Variable*, $\sigma(x) = \sigma_I(x)$. For a query result variable $y \in$ *Variable*, table $t \in$ *Table*, and column $c \in$ *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 $Q \in$ Query, $SQL_\sigma(Q)$ denotes the concrete query Q in SQL syntax:

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

where $C, C_1, C_2 \in$ Col, $E, E_1, E_2 \in$ Expr, $O \in$ Orig, and $y \in$ *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 ($SQL_\sigma(Q)$), 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 $\sigma[Q.y \mapsto \emptyset]$.

$Print_\sigma(Q)$ denotes the output from Q: if Q is of the form

$$\text{“}y \leftarrow \texttt{select}\ \overline{C}\ \texttt{where}\ E\ ;\ \texttt{print}\ \overline{O}\text{”}$$

then $Print_\sigma(Q) = \overline{\sigma[y \mapsto \sigma(Q)](O)}$, where $C \in \text{Col}$, $E \in \text{Expr}$, and $O \in \text{Orig}$.

Example 3.4.5. Following the notation in Example 3.4.2, 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 3-8, that is,

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

We have concrete queries: [3]

$$SQL_{\sigma_1}(Q_1) = \texttt{SELECT * FROM student WHERE id = '0'},$$
$$SQL_{\sigma_2}(Q_1) = \texttt{SELECT * FROM student WHERE id = '5'}.$$

Moreover, $\sigma_1(Q_1) = \emptyset$ and $|\sigma_1(Q_1)| = 0$, consistent with the first example execution

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

(Figure 3-5a). Also, $\sigma_2(Q_1)$ contains the row in the `student` table:

$$\sigma_2(Q_1) = (\{\texttt{student.id} : \texttt{'5'}, \texttt{student.password} : \texttt{'2'}, \texttt{student.firstname} : \texttt{'3'}, \texttt{student.lastname} : \texttt{'4'}\}).$$

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 3-7a).

Definition 3.4.6. 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 3-16a). A concrete trace consists of the intercepted SQL traffic, specifically, the queries CQuery* and corresponding retrieved rows CData*. Clearly for any query $Q \in \text{Query}$, expression $E \in \text{Expr}$, and context $\sigma \in \mathit{Context}$, we have $\mathit{SQL}_\sigma(Q) \in \text{CQuery}$ and $\mathit{SQL}_\sigma(E) \in \text{CExpr}$.

Figure 3-17 presents the rules for executing a program to obtain a concrete trace. We shall write $[\begin{smallmatrix} q \\ d \end{smallmatrix}]$ for the concrete trace $(q\ d) \in \text{CTrace}$. 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 $\mathit{Print}.(\cdot)$. 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 3.4.7. Following the notation in Example 3.4.2 and Example 3.4.5, let Q_2 be the second inferred query in Figure 3-10, that is,

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

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:

$$\sigma_1(P') = \left[\begin{array}{c} \texttt{SELECT * FROM student WHERE id = '0'} \\ \emptyset \end{array} \right].$$

Executing P' in σ_2 would produce concrete trace:

$$\sigma_2(P') = \left[\begin{array}{c} \texttt{SELECT * FROM student WHERE id = '5'}, \quad \texttt{SELECT * FROM student WHERE id = '5'} \wedge \texttt{password = '6'} \\ (\{\texttt{student.id:'5'},\texttt{student.password:'2'},\texttt{student.firstname:'3'},\texttt{student.lastname:'4'}\}), \quad \emptyset \end{array} \right].$$

These two concrete traces are consistent with the two example traces in Figure 3-5a and Figure 3-7a, respectively. So far, the hypothetical program P' is consistent with the observed behavior of the example program (Section 3.1). However, the third context in the example would cause P' to behave inconsistently (Figure 3-9).

Definition 3.4.8. $\boxed{P}$ denotes the black box executable of a program $P \in$ Prog, i.e., executing $\boxed{P}$ in context $\sigma \in$ *Context* produces the concrete trace $\sigma(P)$. Note that KONURE does not access the source code of P when it executes $\boxed{P}$.

Definition 3.4.9. 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 3-16b). 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 unambiguous origin location (Sec-

tion 3.6). With the origin locations inferred, it is straightforward to rewrite the trace syntax as an abstract trace.

Example 3.4.10. Following the notation in Example 3.4.7, the abstract trace for $\sigma_1(P')$ is the same as Figure 3-5b, with a row count 0. The abstract trace for $\sigma_2(P')$ is the same as Figure 3-7b, with row counts $1, 0$.

Definition 3.4.11. 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 3.4.12. Following the notation in Example 3.4.7, 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 3.4.13. A *loop layout tree* for a program $P \in \text{Prog}$ is a tree that represents information about the execution of loops (Figure 3-18). 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 3.4.14. Following the notation in Example 3.4.7, 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 3-11, 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 3-11. 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 3.4.15. 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 $Q \in$ 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 3.4.16. Following the notation in Example 3.4.14, 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:

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

Definition 3.4.17. 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 $Q_1, \ldots, Q_n \in$ 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 3.4.18. In Section 3.1, the first execution does not impose any path constraints,

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

Following the notation in Example 3.4.14, 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 3.4.19. 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 3.4.20. A context $\sigma \in$ *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 $\sigma_1, \ldots, \sigma_n \in$ *Context* are updated according to the 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 $\sigma \in$ *Context* always satisfies the trivial path constraint $W = \text{Nil}$.

Example 3.4.21. Following the notation in Example 3.4.2, Example 3.4.14, and Example 3.4.18, 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 3.1 chooses contexts to infer each production.

Definition 3.4.22. Origin locations $O_1, O_2 \in$ Orig are equivalent with respect to path constraint W, denoted as $O_1 \equiv_W O_2$, if for any context $\sigma \in$ *Context* that satisfies W, O_1, O_2 hold the same values in the context after updating σ with W.

Example 3.4.23. Following the notation in Example 3.4.5, Example 3.4.7, and Example 3.4.18, 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 3.4.24. Expressions $E_1, E_2 \in$ 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 3-19).

Queries $Q_1, Q_2 \in$ 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.

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 3.4.25. Following the notation in Example 3.4.7 and Example 3.4.18, let Q_2' be an alternative second inferred query in Figure 3-10, that is,

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

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

Definition 3.4.26. 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:

$$
\begin{aligned}
& t \sim \mathrm{Nil} = \texttt{true} \\
& t \sim (\langle Q_1', r_1', s_1' \rangle, \ldots, \langle Q_m', r_m', s_m' \rangle) \\
& \qquad = m \leq n \wedge \big(\forall i = 1, \ldots, m : r_i \simeq r_i' \wedge Q_i \doteq_{W_i, Y_i, Y_i'} Q_i'\big)
\end{aligned}
$$

where $W_i = \big(\langle Q_1', r_1', s_1' \rangle, \ldots, \langle Q_{i-1}', r_{i-1}', s_{i-1}' \rangle\big)$ 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 $Y_i' = \big(Q_1'.y, \ldots, Q_{i-1}'.y\big)$ is the list of variables defined by the queries in W_i.

Example 3.4.27. Following the notation in Example 3.4.16 and Example 3.4.18, we have $t_1 \sim W_1$, $t_2 \sim W_1$, $t_{31} \sim W_1$, $t_{32} \sim W_1$, $t_1 \not\sim W_2$, $t_2 \sim W_2$, $t_{31} \sim W_2$, $t_{32} \sim W_2$, $t_1 \not\sim W_3$, $t_2 \not\sim W_3$, $t_{31} \sim W_3$, $t_{32} \sim W_3$, $t_1 \not\sim W_4$, $t_2 \not\sim W_4$, $t_{31} \sim W_4$, $t_{32} \sim W_4$, $t_1 \not\sim W_5$, and $t_2 \not\sim W_5$.

If we additionally have, for example, $r_{41} = 0$, $r_{42} = 1$, and $r_{52} = 3$, then $t_{31} \not\sim W_5$ and $t_{32} \sim W_5$.

Algorithm 1 Infer an executable program

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

```
1: procedure INFER(P)
2:     σ ← Database empty, input parameters distinct
3:     t ← GETTRACE(P, Nil, σ)
4:     return INFERPROG(P, Nil, t)
5: end procedure
```

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.

```
1: procedure GETTRACE(P, W, σ)
2:     e ← EXECUTE(P, σ)
3:     l ← DETECTLOOPS(e)
4:     t ← MATCHPATH(l, W)
5:     return t
6: end procedure
```

3.4.2 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 3.3). 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.

Example 3.4.28. In Section 3.1, 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 3.4.2. The resulting trace (Figure 3-5) is converted into an annotated trace, variable t, which equals t_1 in Example 3.4.16.

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 3-3). 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 3.4.29. In Section 3.1, when Konure executes the program for the first time, it invokes Execute with context σ_1 (Example 3.4.2). Execute configures the database to empty and runs the program with inputs `'0'` and `'1'`. This execution results in the first concrete trace (Figure 3-5a) which equals $\sigma_1(P')$ in Example 3.4.7. Execute converts the concrete trace into an abstract trace, described in Example 3.4.10, and then into a list of query-result pairs that equals e_1 in Example 3.4.12.

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 3.4.30. Following the notation in Example 3.4.12 and Example 3.4.14, 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 3.4.14), 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$(e_3) = l_3$.

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 3.4.31. Following the notation in Example 3.4.14 and Example 3.4.16, we have

$$\begin{aligned}
&\text{GetAnnotatedTrace}(l_1) = \{t_1\},\\
&\text{GetAnnotatedTrace}(l_2) = \{t_2\},\\
&\text{GetAnnotatedTrace}(l_3) = \{t_{31}, t_{32}\}.
\end{aligned}$$

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.

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), ..., (Q_n, r_n) ← e
    a ← empty list
    for j ← 2, 3, ..., n do                                   ▷ Identify repetitions
        if π_S Q_j = π_S Q_2 then
            Append j to a
        end if
    end for
    if r_1 ≤ 1 or r_1 ≠ len(a) then   ▷ Did not find repetitions caused by any loops that
    iterate over Q_1
        e' ← (Q_2, r_2), ..., (Q_n, r_n)
        l ← DetectLoops(e')
        return (Q_1, r_1) ↘ l
    else                                                      ▷ Found a loop that iterates over Q_1
        Append n + 1 to a
        for j ← 1, 2, ..., r_1 do
            b ← a[j]
            c ← a[j + 1] − 1
            e' ← (Q_b, r_b), ..., (Q_c, r_c)
            l_j ← DetectLoops(e')
        end for
        return (Q_1, r_1) ↻ (l_1, ..., l_{r_1})
    end if
end procedure
```

Following the notation in Example 3.4.18, we have

$$\textsc{MatchPath}(l_1, W_1) = t_1,$$

$$\textsc{MatchPath}(l_2, W_2) = t_2,$$

$$\textsc{MatchPath}(l_3, W_3) = t_{31}$$

(or t_{32}, depending on the order in which MatchPath enumerates the traces returned from GetAnnotatedTrace), and

$$\textsc{MatchPath}(l_3, W_4) = t_{31} \quad (\text{or } t_{32}).$$

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 ∼ 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) ↘ l′ then
        return { ⟨Q, r, NotLoop⟩ @ t | t ∈ GetAnnotatedTrace(l′) }
    else if l = (Q, r) ↻ (l′_1, l′_2, ..., l′_r) then          ▷ r ≥ 2
        return ∪_{i=1}^{r} { ⟨Q, r, i⟩ @ t | t ∈ GetAnnotatedTrace(l′_i) }
    end if
end procedure
```

If we additionally have, for example, $r_{41} = 0$, $r_{42} = 1$, and $r_{52} = 3$, then $\textsc{MatchPath}(l_3, W_5) = t_{32}$. In this case t_{31} can no longer be returned, because $t_{31} \not\sim W_5$ (Example 3.4.27).

MakePathConstraint: The MakePathConstraint procedure takes an annotated trace prefix t, a subsequent query $Q \in \text{Query}$, 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 $\langle Q_i, r_i, \lambda_i \rangle$ in

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([P], W)
    σ ← Solve(W)
    if σ = Unsat then
        return false, Nil
    else
        t ← GetTrace([P], W, σ)
        return true, t
    end if
end procedure
```

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 $s'_i = \texttt{true}$ if an 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 $r' = \begin{cases} (= 0) & \text{if } r = 0 \\ (\geq 1) & \text{if } r = 1 \\ (\geq 2) & \text{if } r = 2 \end{cases}$.

Example 3.4.32. Following the notation in Example 3.4.14 and Example 3.4.18, 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 3.4.33. Consider the first execution of the example program in Section 3.1. INFER invokes GETTRACE with context σ_1 (Example 3.4.28). The initial path constraint is $W_1 = \text{Nil}$ (Example 3.4.18). GETTRACE invokes EXECUTE with σ_1, resulting in the list of query-result pairs e_1 (Example 3.4.29). Recall from Example 3.4.30 that $\text{DETECTLOOPS}(e_1) = l_1$. Recall from Example 3.4.31 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 3.4.16). INFERPROG invokes MAKEPATHCONSTRAINT three times, constructing three different path constraints. The first path constraint

specifies that Q_1 retrieves zero rows:

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

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

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

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

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

InferProg then invokes SolveAndGetTrace to determine if these path constraints are satisfiable and, if so, obtain the corresponding annotated traces. In the example (Section 3.1), the first path constraint results in the annotated trace t_1. The second path constraint results in the annotated trace t_2 (Example 3.4.16). The third path constraint is not satisfiable. Based on these results, InferProg applies the "Prog $\coloneqq$ 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 $\coloneqq \epsilon$, Prog $\coloneqq$ Seq, Prog $\coloneqq$ If, and Prog $\coloneqq$ For (Figure 3-6). 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 ($s_1 \mathbin{@} s_2$), 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 Chapter 4.

3.5 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 [54, 160, 155, 156, 83, 149], 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.

3.6 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 $y.\text{Col}$ in P. Concrete traces contain intercepted

```
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)
```

Figure 3-13: KONURE infers the example command and regenerates code in Python

$$
\begin{array}{lcl}
\text{SProg} & := & \epsilon \mid \text{SSeq} \mid \text{SIf} \mid \text{SFor} \\
\text{SSeq} & := & \text{SQuery SProg} \\
\text{SIf} & := & \texttt{if}\ \text{SQuery}\ \texttt{then}\ \text{SProg}\ \texttt{else}\ \text{SProg} \\
\text{SFor} & := & \texttt{for}\ \text{SQuery}\ \texttt{do}\ \text{SProg}\ \texttt{else}\ \text{SProg} \\
\text{SQuery} & := & \diamond \leftarrow \texttt{select}\ \text{SCol}^{+}\ \texttt{where}\ \text{SExpr}\ ;\ \texttt{print}\ \diamond \\
\text{SExpr} & := & \texttt{true} \mid \text{SExpr} \wedge \text{SExpr} \mid \text{SCol} = \text{SCol} \mid \text{SCol} = \diamond \\
\text{SCol} & := & t.c
\end{array}
$$

$$t \in \mathit{Table}, \quad c \in \mathit{Column}, \quad \diamond \text{ is a placeholder}$$

Figure 3-14: Grammar for skeleton programs ($\mathcal{S}$)

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

$$P, P_1, P_2 \in \text{Prog}, \quad Q \in \text{Query}, \quad C, C_1, C_2 \in \text{Col}, \quad E, E_1, E_2 \in \text{Expr}, \quad O \in \text{Orig}, \quad y \in \mathit{Variable}$$

Figure 3-15: Calculating the skeleton of a program

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

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

(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.

Figure 3-16: Grammars for concrete and abstract traces.

$$\frac{}{\sigma(\epsilon) = \left[\begin{smallmatrix}\text{Nil}\\ \text{Nil}\end{smallmatrix}\right]} \quad \text{(epsilon)}$$

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

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

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

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

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

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

Figure 3-17: Semantics for executing a program using a context to obtain a concrete trace

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

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

Figure 3-18: Grammar for loop layout trees

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

$$\frac{O \equiv_W O'}{(C = O) \doteq_W (C = O')} \quad \text{(orig)} \qquad \frac{E_1 \doteq_W E_1' \qquad E_2 \doteq_W E_2'}{(E_1 \wedge E_2) \doteq_W (E_1' \wedge E_2')} \quad \text{(and)}$$

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

Figure 3-19: Check if two expressions are identical except for equivalent variables with respect to a path constraint

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, s_1, s_2)
    if s_2 = Nil then return ϵ                                  ▷ Prog := ϵ
    end if
    k ← The length of s_1
    Q ← The first query in s_2
    for i = 0, 1, 2 do
        W_i ← MakePathConstraint(s_1, 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
```

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.

Chapter 4

Soundness Proof of KONURE

In this chapter, 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 4.2 and 4.3 elaborate on the transformation and the functions that are used to define $\mathcal{K}$ in Section 3.3 (Definition 3.3.4). Section 4.4 proves Theorem 1. Section 4.5 proves Theorem 2. Section 4.6 proves Theorem 3 and Theorem 4. Section 4.7 proves Theorem 5.

4.1 Soundness Proof Overview

Definition 4.1.1. Programs $P_1, P_2 \in \text{Prog}$ 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 4-1).

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 4-1.

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

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

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

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

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

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

Figure 4-1: Check if two programs are identical except for equivalent variables

Definition 4.1.2. For a program $P \in \text{Prog}$ 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 4-2 defines this evaluation.

$\sigma \vdash P \Downarrow_{\text{loops}} l$ denotes evaluating P in σ to obtain a loop layout tree l (see Definition 3.4.13). Figure 4-3 defines this evaluation.

Definition 4.1.3. For programs $P, P' \in \text{Prog}$ 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 4-4 defines this traversal.

For loop layout trees l, l' and annotated trace t, we use the notation $l \overset{t}{\hookrightarrow} l'$ 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 4-5 defines this traversal.

Definition 4.1.4. A path constraint W is *derived from* a program $P \in \text{Prog}$ if one of the following holds:

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

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

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

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

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

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

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

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

Figure 4-2: Semantics for executing a program using a context to directly obtain a list of query-result pairs

2. $W = \langle Q', r', s' \rangle$, $P \neq \epsilon$, and $Q' \doteq_{\text{Nil,Nil,Nil}} \mathcal{F}(P)$, where $\mathcal{F}(P)$ is the first query in P. We formally define the function $\mathcal{F}(\cdot)$ in Section 4.3.

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 $W' = \left(\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \left\langle Q'_{m-1}, r'_{m-1}, s'_{m-1} \right\rangle, \langle Q'_m, r', s'_m \rangle\right)$ 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 4.1.5. 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:

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

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

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

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

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

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

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

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

Figure 4-3: Semantics for executing a program using a context to obtain a loop layout tree

$$\|\epsilon\| = 1$$

$$\|Q\ P\| = 1 + \|P\|$$

$$\|\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2\| = 1 + \|P_1\| + \|P_2\|$$

$$\|\texttt{for}\ Q\ \texttt{do}\ P_1\ \texttt{else}\ P_2\| = 1 + \|P_1\| + \|P_2\|$$

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

Proposition 4.1.6 (Solver)**.** For any path constraint W, the procedure $\textsc{Solve}(W)$ returns a context $\sigma \in \textit{Context}$ if and only if W is satisfiable.

Rationale. The path constraint solver outlined in Section 3.5 asks the SMT solver a question that is equisatisfiable as the existence of a satisfying context. Since the

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

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

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

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

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

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

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

Figure 4-4: Traverse a program by following an annotated trace, to obtain a subprogram

logical formulas are quantifier-free and involve only equality checks, their satisfiability is decidable, for example by first converting the formulas to disjunctive normal form (DNF) and then checking each disjunct with a congruence closure algorithm [37]. Our Konure prototype uses an off-the-shelf SMT solver, Z3 [61], that works efficiently in practice.

□

Proposition 4.1.7 (Disambiguation)**.** For any program $P \in \mathcal{K}$ and context $\sigma \in$ *Context*, if $\sigma \vdash P \Downarrow_{\text{exec}} e$, 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 3.6) asks the SMT solver a question that equivalently encodes the relationship between origin locations. By Proposition 4.1.6, we obtain a correct list of query-result pairs after disambiguating the traces obtained from program execution. □

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

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

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

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

Figure 4-5: Traverse a loop layout tree by following an annotated trace, to obtain a subtree

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 4-2.

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

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

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 $\sigma \in \mathit{Context}$ that satisfies W, and annotated trace t, if $t = \textsc{GetTrace}(\boxed{P}, W, \sigma)$ then there exists a loop layout tree l' such that:

1. $\sigma \vdash P \Downarrow_{\text{loops}} l'$,
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 4.5. □

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 $P \in \text{Prog}$ 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 4.6. □

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 4.6. □

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 $\textsc{InferProg}$ procedure at most $\|P\|$ times.

Proof Sketch. By induction on the derivation of P. We present a full proof in Section 4.7. □

Intuition. Each recursive call to $\textsc{InferProg}$ 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 $\textsc{InferProg}$ 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 3-6).

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 4.8.

4.2 The Trim Transformation

This section presents the transformation that obtains $\widetilde{P}$, which we introduce in Section 3.3 to define the Konure DSL, $\mathcal{K}$ (Definition 3.3.4). Recall from Section 3.3 that, for any program $P \in \text{Prog}$, $\widetilde{P}$ is the program after discarding unreachable code in P. The reachability properties facilitate the soundness proof in Section 4.6 and enable a concise way to characterize complexity in Section 4.7. In Section 4.8 we will extend our soundness results to programs not in $\mathcal{K}$, such as programs in Prog that may contain unreachable code.

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
```

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, $P \in \text{Prog}$, and a path constraint, W. The procedure returns a tuple of two components. The first component is a boolean value which indicates whether the transformation alters P. The second component is the transformed program. If P is empty, 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 3.5) to check reachability.

We show that the TRIM transformation terminates (Theorem 6) with an equivalent program (Theorem 7) with no unreachable code (Proposition 4.2.17).

4.2.1 Termination of TRIM

To show termination, we define a measure of code size and show that the TRIMONCE transformation decreases the code size.

Definition 4.2.1. The *branch complexity tuple* for a program $P \in \text{Prog}$ 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,

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 P_1 then
        (s_1, P'_1) ← TrimOnce(P_1, W)
        return (s_1, Q P'_1)
    else if P = if Q then P_1 else P_2 then
        if Solve(W @ ⟨Q, ≥ 1, false⟩) = Unsat then return (true, Q P_2)
        else if Solve(W @ ⟨Q, = 0, false⟩) = Unsat then return (true, Q P_1)
        end if
        (s_1, P'_1) ← TrimOnce(P_1, W @ ⟨Q, ≥ 1, false⟩)
        (s_2, P'_2) ← TrimOnce(P_2, W @ ⟨Q, = 0, false⟩)
        if P'_1 ≐_{W,Nil,Nil} P'_2 then
            return (true, Q P'_1)
        else
            return (s_1 ∨ s_2, if Q then P'_1 else P'_2)
        end if
    else if P = for Q do P_1 else P_2 then
        if Solve(W @ ⟨Q, ≥ 2, true⟩) = Unsat then
            return (true, if Q then P_1 else P_2)
        end if
        (s_1, P'_1) ← TrimOnce(P_1, W @ ⟨Q, ≥ 1, true⟩)
        if P'_1 = ϵ then
            return (true, if Q then ϵ else P_2)
        end if
        if Solve(W @ ⟨Q, = 0, true⟩) = Unsat then
            s_2 ← (P_2 ≠ ϵ)
            P'_2 ← ϵ
        else
            (s_2, P'_2) ← TrimOnce(P_2, W @ ⟨Q, = 0, true⟩)
        end if
        return (s_1 ∨ s_2, for Q do P'_1 else P'_2)
    end if
end procedure
```

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:

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

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

Definition 4.2.2. 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 4.2.3. 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 4.2.4. *For any program $P \in Prog$ and path constraint W, if* $\textsc{TrimOnce}(P, W) = (\texttt{false}, P')$ *then* $P' = P$.

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

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

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

<u>Case 2:</u> 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

$$\textsc{TrimOnce}(P, W) = (\texttt{false}, P'),$$

$s_1 = \texttt{false}$ and $P' = Q\ P_1'$. By the induction hypothesis, if $s_1 = \texttt{false}$ then $P_1' = P_1$. Hence, $P' = Q\ P_1' = Q\ P_1 = P$.

<u>Case 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.

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

$$\textsc{TrimOnce}(P, W) = (\texttt{false}, P'),$$

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, $P' = \texttt{if}\ Q\ \texttt{then}\ P_1'\ \texttt{else}\ P_2' = \texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2 = P$.

<u>Case 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 3.

□

Lemma 4.2.5. *For any program $P \in$ Prog and path constraint W, if* $\textsc{TrimOnce}(P, W) = (\texttt{true}, P')$ *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 $\textsc{TrimOnce}(P, W) = (\texttt{true}, P')$.

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

$$\textsc{TrimOnce}(P, W) = (\texttt{true}, P'),$$

$s_1 = \texttt{true}$ 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 4.2.1 and Definition 4.2.2, $\mathcal{B}(Q\ P_1') < \mathcal{B}(Q\ P_1)$. 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 4.2.1, $\mathcal{B}(Q\ P_2) = (f_2, i_2, 1 + s_2)$. 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 4.2.2, $(f_2, i_2, 1 + s_2) < (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2)$. Since $\textsc{TrimOnce}(P, W) = (\texttt{true}, P')$, $P' = Q\ P_2$. Hence, $\mathcal{B}(P') = \mathcal{B}(Q\ P_2) < \mathcal{B}(\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2) = \mathcal{B}(P)$.

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

<u>Case 3.3:</u> 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 4.2.4, 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 4.2.1, $\mathcal{B}(Q\ P_1') = (f_1', i_1', 1 + s_1')$. 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 4.2.2, $(f_1', i_1', 1 + s_1') < (f_1 + f_2, 1 + i_1 + i_2, s_1 + s_2)$. 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 4.2.1 and Definition 4.2.2, $\mathcal{B}(\texttt{if}\ Q\ \texttt{then}\ P_1'\ \texttt{else}\ P_2') < \mathcal{B}(\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2)$. Hence, $\mathcal{B}(P') < \mathcal{B}(P)$.

<u>Case 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 3.

□

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

Proof. By Lemma 4.2.5, $\mathcal{B}(P') < \mathcal{B}(P)$ 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 4.2.3, 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 4.2.5. Execution enters the branch on line 4. The algorithm then terminates. □

Proposition 4.2.6. For any program $P \in$ Prog, TRIM(TRIM(P)) = TRIM(P).

Proof. For any program $P_0 \in$ Prog, let $P_1 =$ TRIM(P_0). In the last iteration of the loop in Algorithm 7, variable $s = \texttt{false}$ on line 3. By Lemma 4.2.4, variable $P' = P$ at this time. Since the algorithm returns P_1, we have TRIMONCE(P_1, Nil) = ($\texttt{false}, P_1$). Let $P_2 =$ 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, TRIM(TRIM(P_0)) = TRIM(P_0). □

4.2.2 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 4.2.7. 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 4.2.8. For any programs $P_1, P_2 \in \text{Prog}$, if $P_1 \equiv P_2$ then $\textsc{Infer}(\boxed{P_1}) = \textsc{Infer}(\boxed{P_2})$.

Proof. By Definition 4.2.7, $\sigma(P_1) = \sigma(P_2)$ for any context $\sigma \in \mathit{Context}$. By Definition 3.4.8, 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 4.2.9. 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 $\sigma \in \mathit{Context}$ 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 4.2.10. For any programs $P_1, P_2 \in \text{Prog}$ and list of variables $Y \in \overline{\mathit{Variable}}$, $P_1 \equiv_{\text{Nil,Nil,Nil}} P_2$ if and only if $P_1 \equiv_{\text{Nil},Y,Y} P_2$.

Proof. By definition. □

Proposition 4.2.11. For any programs $P_1, P_2 \in \text{Prog}$, $P_1 \equiv_{\text{Nil,Nil,Nil}} P_2$ if and only if $P_1 \equiv P_2$.

Proof. By definition. □

Proposition 4.2.12. For any programs $P_1, P_2 \in \text{Prog}$, path constraint W, and lists of variables $Y_1, Y_2 \in \overline{\mathit{Variable}}$, 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 3.4.24 and Figure 4-1. □

Proposition 4.2.13. For any programs $P_1, P_2 \in \text{Prog}$, if $P_1 \doteq P_2$ then $P_1 \equiv P_2$.

Proof. By Proposition 4.2.11 and Proposition 4.2.12. □

Lemma 4.2.14. *For any program $P \in \mathit{Prog}$ and path constraint W, if* $\textsc{TrimOnce}(P, W) = (s, P')$ *then* $P' \equiv_{W,\mathit{Nil},\mathit{Nil}} P$.

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 4.2.9, $P' \equiv_{W,\text{Nil},\text{Nil}} P$.

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 $\sigma \in \mathit{Context}$ that satisfies W, let $\sigma_1 = \sigma[Q.y \mapsto \sigma(Q)]$. Since σ_1 only adds the mapping of a new variable $Q.y$ which does not appear in W, σ_1 also satisfies W. By Definition 4.2.9, $\sigma_1(P_1') = \sigma_1(P_1)$. By Figure 3-17, $\sigma(Q\ P_1') = \sigma(Q\ P_1)$. By Definition 4.2.9, $Q\ P_1' \equiv_{W,\text{Nil},\text{Nil}} Q\ P_1$. 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 "$\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2$", 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.

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 4.1.6, for any context $\sigma \in \mathit{Context}$ that satisfies W, $\sigma(Q) = \emptyset$ because $|\sigma(Q)| \geq 1$ is impossible. Hence, $\sigma[Q.y \mapsto \sigma(Q)] = \sigma$. By Figure 3-17, $\sigma(Q\ P_2) = \sigma(\texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2)$. By Definition 4.2.9, $Q\ P_2 \equiv_{W,\text{Nil},\text{Nil}} \texttt{if}\ Q\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. Since $\textsc{TrimOnce}(P, W) = (s, P')$, $P' = Q\ P_2$. Hence, $P' \equiv_{W,\text{Nil},\text{Nil}} P$.

Case 3.2: $\textsc{Solve}(W \mathbin{@} \langle Q, = 0, \texttt{false}\rangle)$ is unsatisfiable.

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

Case 3.3: Both $\textsc{Solve}(W \mathbin{@} \langle Q, \geq 1, \texttt{false}\rangle)$ and $\textsc{Solve}(W \mathbin{@} \langle Q, = 0, \texttt{false}\rangle)$ are satisfiable.

Execution continues after line 9. By the induction hypothesis, variables P_1' and P_2' satisfy

$$P_1' \equiv_{W \,@\, \langle Q, \geq 1, \texttt{false} \rangle, \text{Nil}, \text{Nil}} P_1,$$
$$P_2' \equiv_{W \,@\, \langle Q, = 0, \texttt{false} \rangle, \text{Nil}, \text{Nil}} P_2.$$

For any context $\sigma \in \mathit{Context}$ that satisfies W, only one of $|\sigma(Q)| = 0$ and $|\sigma(Q)| \geq 1$ holds.

<u>Case 3.3.1:</u> If $|\sigma(Q)| = 0$, σ satisfies $W \,@\, \langle Q, = 0, \texttt{false} \rangle$. By Definition 4.2.9, $\sigma(P_2') = \sigma(P_2)$. Since $\sigma(Q) = \emptyset$, $\sigma[Q.y \mapsto \sigma(Q)] = \sigma$. By Figure 3-17,

$$\sigma(\texttt{if } Q \texttt{ then } P_1' \texttt{ else } P_2') = \sigma(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2).$$

<u>Case 3.3.2:</u> If $|\sigma(Q)| \geq 1$, 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 $Q.y$ which does not appear in W, σ_1 also satisfies the path constraint $W \,@\, \langle Q, \geq 1, \texttt{false} \rangle$. By Definition 4.2.9, $\sigma_1(P_1') = \sigma_1(P_1)$. By Figure 3-17, $\sigma(\texttt{if } Q \texttt{ then } P_1' \texttt{ else } P_2') = \sigma(\texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2)$.

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 4.2.9, $\texttt{if } Q \texttt{ then } P_1' \texttt{ else } P_2' \equiv_{W,\text{Nil},\text{Nil}} \texttt{if } Q \texttt{ then } P_1 \texttt{ else } P_2$.

Next, consider if P_1' and P_2' are identical except for equivalent variables.

<u>Case 3.3.1:</u> If $P_1' \doteq_{W,\text{Nil},\text{Nil}} P_2'$, execution enters the branch on line 12. By Proposition 4.2.12, $P_1' \equiv_{W,\text{Nil},\text{Nil}} P_2'$. By Defini-

tion 4.2.9 and Figure 3-17, 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 4.2.9, $P' \equiv_{W,\text{Nil},\text{Nil}} P$.

Case 3.3.2: If $P_1' \not\equiv_{W,\text{Nil},\text{Nil}} P_2'$, 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 4.2.15. *For any program $P \in$ Prog, if $\textsc{TrimOnce}(P, Nil) = (s, P')$ then $P' \equiv P$.*

Proof. By Lemma 4.2.14, $P' \equiv_{\text{Nil},\text{Nil},\text{Nil}} P$. By Proposition 4.2.11, $P' \equiv P$. □

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

Proof. By Lemma 4.2.15, $P' \equiv P$ on line 3 in each iteration of the loop. By Theorem 6, the loop terminates. By Definition 4.2.7, 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 4.2.16. For any program $P \in$ Prog and path constraint W, if

$$\textsc{TrimOnce}(P, W) = (\texttt{false}, P')$$

then the following hold for P':

- For any query $Q \in$ Query in P', there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(P')$.
- For any `if`-construct "`if` $Q \ldots$" in P', there exists a context $\sigma \in$ *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 4.1.6. □

Proposition 4.2.17 (Reachability)**.** For any program $P \in$ Prog, the following hold:

- For any query $Q \in$ Query in $\textsc{Trim}(P)$, there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(\textsc{Trim}(P))$.
- For any `if`-construct "`if` $Q \ldots$" in $\textsc{Trim}(P)$, there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(\textsc{Trim}(P))$ and the corresponding row count $r \geq 1$.
- For any `if`-construct "`if` $Q \ldots$" in $\textsc{Trim}(P)$, there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(\textsc{Trim}(P))$ and the corresponding row count $r = 0$.
- For any `for`-construct "`for` $Q \ldots$" in $\textsc{Trim}(P)$, there exists a context $\sigma \in$ *Context* such that Q is used while evaluating $\sigma(\textsc{Trim}(P))$ 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 4.2.16. □

4.3 Source Code Characteristics

This section defines the functions $\mathcal{T}(\cdot)$, $\mathcal{R}(\cdot)$, and $\mathcal{D}(\cdot)$, which we introduce in Section 3.3 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 4.3.1. For any program $P \in \text{Prog}$, 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 $P, P_1, P_2 \in \text{Prog}$ and $Q \in \text{Query}$.

Definition 4.3.2. Let SQuery be the set of skeleton queries (Appendix **??**). For any program $P \in \text{Prog}$ and $q \in \text{SQuery} \cup \{\,\text{Nil}\,\}$, function $\mathcal{C}(q, P)$ returns the number of queries in P that share the skeleton q:

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

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

Definition 4.3.3. For any program $P \in \text{Prog}$, 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 $P, P_1, P_2 \in \text{Prog}$ and $Q \in \text{Query}$.

Definition 4.3.4. For any program $P \in \text{Prog}$ and context $\sigma \in \mathit{Context}$, $\mathcal{T}_\sigma(P)$ 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 4.3.5. 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 4.3.6. 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.

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

Lemma 4.4.1. *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 4-2, $e = \text{Nil}$. 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 4-2, $Q_1 = Q$ and $r_1 = |\sigma(Q)| = |\sigma(Q_1)|$. By Definition 4.3.1, $\mathcal{F}(P) = Q = Q_1$. By Definition 4.3.4, 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 4.3.5, $Q_y.y \in \mathcal{T}(P_0)$.

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 4.4.2. *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* $\textsc{DetectLoopsAux}(e_0, P_0, \sigma_0)$*, if the parameters of a recursive call* $\textsc{DetectLoopsAux}(e, P, \sigma)$ *satisfy* $\sigma \vdash P \Downarrow_{exec} e$ *and* $\sigma \vdash P \Downarrow_{loops} l$ *then* $\textsc{DetectLoopsAux}(e, P, \sigma) = l$.

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 4-2, $e = \text{Nil}$. Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 4-3, $l = \text{Nil}$. By Algorithm 9, $\textsc{DetectLoopsAux}(\text{Nil}, P, \sigma) = \text{Nil}$.

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 4-2, there exists a list of query-result pairs e_1 such that $e = (Q, r) \mathbin{@} e_1$ and $\sigma_1 \vdash P_1 \Downarrow_{\text{exec}} e_1$. By Lemma 4.4.1, $e = (Q_1, r_1) \mathbin{@} e_1$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 4-3, 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 4.4.1, $l = (Q_1, r_1) \searrow l_1$.

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

Case 2.2: $r \geq 2$. By Lemma 4.4.1, $Q.y \in \mathcal{T}(P_0)$. By Definition 3.3.4, $Q.y \notin \mathcal{R}(P_0)$. By Definition 4.3.3, $\mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) \leq 1$. By Figure 4-2, $\pi_{\mathcal{S}}\mathcal{F}(P_1)$ 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 4.4.1, variable $\sigma' = sigma_1$. Also, variables $e' = e_1$ and $P' = P_1$. Algorithm 9 recursively calls $\textsc{DetectLoopsAux}(e_1, P_1, \sigma_1)$ on line 20, which returns l_1 by the induction hypothesis. Hence

$$\begin{aligned}
\textsc{DetectLoopsAux}(e, P, \sigma) &= (Q_1, r_1) \searrow \textsc{DetectLoopsAux}(e_1, P_1, \sigma_1) \\
&= (Q, r) \searrow l_1 \\
&= l.
\end{aligned}$$

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 4-2, there exists a list of query-result pairs e_2 such that $e = (Q, 0) \mathbin{@} e_2$ and $\sigma \vdash P_2 \Downarrow_{\text{exec}} e_2$. By Lemma 4.4.1, $e = (Q_1, r_1) \mathbin{@} e_2$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 4-3, 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 4.4.1, $l = (Q_1, r_1) \searrow l_2$.

By Lemma 4.4.1, $r_1 = |\sigma(Q)| = 0$. In Algorithm 9, execution enters the branch on line 12. By Definition 3.4.4, variable $\sigma' = \sigma$. Also, variables $e' = e_2$ and $P' = P_2$. Algorithm 9 recursively calls $\textsc{DetectLoopsAux}(e_2, P_2, \sigma)$ on line 20, which returns l_2 by the induction hypothesis. Hence $\textsc{DetectLoopsAux}(e, P, \sigma) = (Q_1, r_1) \searrow \textsc{DetectLoopsAux}(e_2, P_2, \sigma) = (Q, 0) \searrow l_2 = l$.

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 4-2, there exists lists of query-result pairs $e_1, \ldots, e_r$ such that $e = (Q, r) \mathbin{@} e_1 \mathbin{@} \ldots \mathbin{@} e_r$ and $\sigma_i \vdash P_1 \Downarrow_{\text{exec}} e_i$ for each $i = 1, \ldots, r$. By Lemma 4.4.1, $r = r_1$ and $e = (Q_1, r_1) \mathbin{@} e_1 \mathbin{@} \ldots \mathbin{@} e_{r_1}$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 4-3, 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 4.4.1, $Q.y \in \mathcal{T}(P_0)$. By Definition 3.3.4, $Q.y \notin \mathcal{R}(P_0)$. By Definition 4.3.3, $\mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) \leq 1$.

By Definition 3.3.4 and Algorithm 8, $P_1 \neq \epsilon$. By Definition 4.3.2, $\mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) \geq 1$. Hence, $\mathcal{C}(\pi_{\mathcal{S}}\mathcal{F}(P_1), P_1) = 1$.

By Figure 4-2, $\pi_{\mathcal{S}}\mathcal{F}(P_1)$ 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 i-th 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 4.4.1, variable $\sigma' = \sigma[Q_1.y \mapsto x_i] = \sigma_i$. Also, variables $e' = e_i$ and $P' = P_1$. Algorithm 9 recursively calls $\textsc{DetectLoopsAux}(e_i, P_1, \sigma_i)$ on line 30, which returns l_i by the induction hypothesis. Hence

$$
\begin{aligned}
& \textsc{DetectLoopsAux}(e, P, \sigma) \\
= \; & (Q_1, r_1) \circlearrowright (\textsc{DetectLoopsAux}(e_1, P_1, \sigma_1), \ldots, \\
& \qquad \textsc{DetectLoopsAux}(e_r, P_1, \sigma_{r_1})) \\
= \; & (Q, r) \circlearrowright (l_1, \ldots, l_r) \\
= \; & l.
\end{aligned}
$$

□

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

Proof. By Lemma 4.4.2, $\textsc{DetectLoopsAux}(e, P, \sigma) = l$. Since Algorithm 9 and Algorithm 3 differ only in the auxiliary variables, $\textsc{DetectLoops}(e) = l$. □

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: $\sigma \in \mathit{Context}$ is an auxiliary variable used only in the soundness proof.
Output: Loop layout tree constructed from e.

```
procedure DetectLoopsAux(e, P, σ)
    if e = Nil then
        return Nil
    end if
    (Q_1, r_1), ..., (Q_n, r_n) ← e
    a ← empty list
    for j ← 2, 3, ..., n do                                   ▷ Identify repetitions
        if π_S Q_j = π_S Q_2 then
            Append j to a
        end if
    end for
    if r_1 ≤ 1 or r_1 ≠ len(a) then   ▷ Did not find repetitions caused by any loops that
    iterate over Q_1
        e' ← (Q_2, r_2), ..., (Q_n, r_n)
        σ' ← σ[Q_1.y ↦ σ(Q_1)]
        if r_1 = 0 then
            P' ← Subprogram of P in the empty branch
        else
            P' ← Subprogram of P in the nonempty branch
        end if
        l ← DetectLoopsAux(e', P', σ')
        return (Q_1, r_1) ↘ l
    else                                          ▷ Found a loop that iterates over Q_1
        Append n + 1 to a
        for j ← 1, 2, ..., r_1 do
            b ← a[j]
            c ← a[j + 1] − 1
            e' ← (Q_b, r_b), ..., (Q_c, r_c)
            σ' ← σ[Q_1.y ↦ σ(Q_1)[j]]
            P' ← Subprogram of P in the nonempty branch
            l_j ← DetectLoopsAux(e', P', σ')
        end for
        return (Q_1, r_1) ↻ (l_1, ..., l_{r_1})
    end if
end procedure
```

4.5 Soundness of GetTrace

We show that the outcome of MatchPath corresponds to a path through the program's abstract syntax tree and corresponds to a path through the loop layout tree (Lemma 4.5.9). 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 4.5.1) and subtrees of the loop layout tree (Section 4.5.2).

4.5.1 Traversing the Program AST

Proposition 4.5.1. For any programs $P_1, P_2, P_3 \in \text{Prog}$ 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 $P_1 \xrightarrow{t_1 \,@\, t_2} P_3$, $P_1 \xrightarrow{t_1} P_2$ may not hold. Consider the following programs:

$$P_1 = Q_1\ Q_2$$
$$P_2 = \texttt{if}\ Q_2\ \texttt{then}\ Q_3\ \texttt{else}\ \epsilon$$
$$P_3 = \epsilon$$

Let $t_1 = \langle Q_1, 0, \text{NotLoop}\rangle$ and $t_2 = \langle Q_2, 0, \text{NotLoop}\rangle$. By Figure 4-4, $P_2 \xrightarrow{t_2} \epsilon$, $P_1 \xrightarrow{t_1 \,@\, t_2} \epsilon$, and $P_1 \xrightarrow{t_1} Q_2$. However, $Q_2 \neq P_2$.

Proposition 4.5.2. For any programs $P_1, P_2 \in \text{Prog}$ and annotated traces t_1, t_2, if $P_1 \xrightarrow{t_1 \,@\, t_2} P_2$ then there exists program $P_3 \in \text{Prog}$ 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 4-4, $P_1 \xrightarrow{\text{Nil}} P_1$.

<u>Case 2:</u> $t_1 = \langle Q', r, \lambda\rangle \,@\, t_1'$. We have $t_1 \,@\, t_2 = \langle Q', r, \lambda\rangle \,@\, t_1' \,@\, t_2$.

Case 2.1: $P_1 = \epsilon$. Since $t_1 \mathbin{@} t_2 \neq \text{Nil}$, it is not possible to have $P_1 \xrightarrow{t_1 \mathbin{@} t_2} P_2$ by Figure 4-4.

Case 2.2: 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 $P_1 \xrightarrow{t_1 \mathbin{@} t_2} P_2$, by Figure 4-4, $Q \doteq Q'$ and $P_1' \xrightarrow{t_1' \mathbin{@} t_2} P_2$. By the induction hypothesis, there exists program $P_3 \in \text{Prog}$ such that $P_1' \xrightarrow{t_1'} P_3$. By Figure 4-4, $P_1 \xrightarrow{\langle Q', r, \lambda \rangle} P_1'$. By Proposition 4.5.1, $P_1 \xrightarrow{t_1} P_3$.

Case 2.3: 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 2.3.1: $r > 0$. The proof is similar to the proof of Case 2.2.

Case 2.3.2: $r = 0$.

Since $P_1 \xrightarrow{t_1 \mathbin{@} t_2} P_2$, by Figure 4-4, $Q \doteq Q'$ and $P_2' \xrightarrow{t_1' \mathbin{@} t_2} P_2$.

By the induction hypothesis, there exists program $P_3 \in \text{Prog}$ such that $P_2' \xrightarrow{t_1'} P_3$. By Figure 4-4, $P_1 \xrightarrow{\langle Q', r, \lambda \rangle} P_2'$. By Proposition 4.5.1, $P_1 \xrightarrow{t_1} P_3$.

Case 2.4: 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 2.3.

□

Proposition 4.5.3. For any program $P \in \text{Prog}$, context $\sigma \in \mathit{Context}$, loop layout tree l such that $\sigma \vdash P \Downarrow_{\text{loops}} l$, and annotated trace $t \in \textsc{GetAnnotatedTrace}(l)$, we have $P \xrightarrow{t} \epsilon$.

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

Case 1: $P = \epsilon$.

By Figure 4-3, $\sigma \vdash P \Downarrow_{\text{loops}} \text{Nil}$. By Algorithm 4,

$$\textsc{GetAnnotatedTrace}(\text{Nil}) = \{\, \text{Nil} \,\}.$$

Hence $t = \text{Nil}$. By Figure 4-4, $\epsilon \xrightarrow{\text{Nil}} \epsilon$.

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 4-3, 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,

$$\begin{aligned} &\textsc{GetAnnotatedTrace}(l) \\ &= \{\, \langle Q, r, \text{NotLoop} \rangle \mathbin{@} t' \mid t' \in \textsc{GetAnnotatedTrace}(l_1) \,\}. \end{aligned}$$

Since $t \in \textsc{GetAnnotatedTrace}(l)$, there exists

$$t' \in \textsc{GetAnnotatedTrace}(l_1)$$

such that $t = \langle Q, r, \text{NotLoop} \rangle \mathbin{@} t'$.

By the induction hypothesis, $P_1 \xrightarrow{t'} \epsilon$. By Figure 4-4, $P \xrightarrow{t} \epsilon$.

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 4-3, 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 $i = 1, \ldots, r$. By Algorithm 4, $\textsc{GetAnnotatedTrace}(l) = \cup_{i=1}^{r} \{ \langle Q, r, i \rangle \mathbin{@} t' \mid t' \in \textsc{GetAnnotatedTrace}(l_i) \}$.

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

By the induction hypothesis, $P_1 \xrightarrow{t'} \epsilon$. By Figure 4-4, $P \xrightarrow{t} \epsilon$.

□

4.5.2 Traversing the Loop Layout Tree

Proposition 4.5.4. For any loop layout trees l_1, l_2, l_3 and annotated traces t_1, t_2:

1. if $l_1 \overset{t_1}{\hookrightarrow} l_2$ and $l_2 \overset{t_2}{\hookrightarrow} l_3$ then $l_1 \xhookrightarrow{t_1 \mathbin{@} t_2} l_3$.
2. if $l_1 \overset{t_1}{\hookrightarrow} l_2$ and $l_1 \xhookrightarrow{t_1 \mathbin{@} t_2} l_3$ then $l_2 \overset{t_2}{\hookrightarrow} l_3$.

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 $l_2 \overset{t_2}{\hookrightarrow} l_3$ and $l_1 \xhookrightarrow{t_1 \mathbin{@} t_2} l_3$, $l_1 \overset{t_1}{\hookrightarrow} l_2$ may not hold. Consider the following loop layout trees:

$$l_1 = (Q_1, 0) \searrow (Q_2, 2) \circlearrowright ((Q_3, 0) \searrow \text{Nil}\,,\ (Q_3, 3) \searrow \text{Nil}\,)$$

$$l_2 = (Q_2, 2) \circlearrowright ((Q_3, 0) \searrow \text{Nil}\,,\ (Q_3, 1) \searrow \text{Nil}\,)$$

$$l_3 = (Q_3, 0) \searrow \text{Nil}$$

Let $t_1 = \langle Q_1, 0, \text{NotLoop} \rangle$ and $t_2 = \langle Q_2, 2, 1 \rangle$. By Figure 4-5, $l_2 \overset{t_2}{\hookrightarrow} l_3$, $l_1 \xhookrightarrow{t_1 \mathbin{@} t_2} l_3$, and $l_1 \overset{t_1}{\hookrightarrow} l_2'$ where $l_2' = (Q_2, 2) \circlearrowright ((Q_3, 0) \searrow \text{Nil}\,,\ (Q_3, 3) \searrow \text{Nil}\,)$. However, $l_2 \neq l_2'$.

Proposition 4.5.5. 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 $l_1 \hookrightarrow^{t_1} l_3$.

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

<u>Case 1:</u> $t_1 = \text{Nil}$. Let $l_3 = l_1$. By Figure 4-5, $l_1 \hookrightarrow^{\text{Nil}} l_1$.

<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> $l_1 = \text{Nil}$. 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 4-5.

<u>Case 2.2:</u> $l_1 = (Q, r) \searrow l_1'$.
Since $l_1 \xhookrightarrow{t_1 \,@\, t_2} l_2$, by Figure 4-5, $Q \doteq Q'$, $\lambda = \text{NotLoop}$, and $l_1' \xhookrightarrow{t_1' \,@\, t_2} l_2$. By the induction hypothesis, there exists loop layout tree l_3 such that $l_1' \hookrightarrow^{t_1'} l_3$. By Figure 4-5, $l_1 \xhookrightarrow{\langle Q', r, \lambda \rangle} l_1'$. By Proposition 4.5.4, $l_1 \hookrightarrow^{t_1} l_3$.

<u>Case 2.3:</u> $l_1 = (Q', r) \circlearrowright (l_1', \ldots, l_r')$.
Since $l_1 \xhookrightarrow{t_1 \,@\, t_2} l_2$, by Figure 4-5, $Q \doteq Q'$, $1 \leq \lambda \leq r$, and $l_\lambda' \xhookrightarrow{t_1' \,@\, t_2} l_2$. By the induction hypothesis, there exists loop layout tree l_3 such that $l_\lambda' \hookrightarrow^{t_1'} l_3$. By Figure 4-5, $l_1 \xhookrightarrow{\langle Q', r, \lambda \rangle} l_\lambda'$. By Proposition 4.5.4, $l_1 \hookrightarrow^{t_1} l_3$.

□

Proposition 4.5.6. For any loop layout tree l and annotated trace t, we have $t \in \textsc{GetAnnotatedTrace}(l)$ if and only if $l \hookrightarrow^{t} \text{Nil}$.

Proof. By induction on the length of t. □

4.5.3 Consistency with Program AST, Path Constraint, and Loop Layout Tree

Lemma 4.5.7. *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 $t \in \textsc{GetAnnotatedTrace}(l)$ such that $t \sim W$.*

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

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

By Figure 4-3, $l = \text{Nil}$. By Algorithm 4, $\textsc{GetAnnotatedTrace}(l) = \{\,\text{Nil}\,\}$.

Let $t = \text{Nil}$, then $t \in \textsc{GetAnnotatedTrace}(l)$.

<u>Case 1.1:</u> $W = \text{Nil}$.

By Definition 3.4.26, $t \sim W$.

<u>Case 1.2:</u> $W = \langle Q', r', s' \rangle$.

By Definition 4.3.1, $\mathcal{F}(P) = \text{Nil}$. By Definition 4.1.4, this case is not possible.

<u>Case 1.3:</u> $W = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$, where $m \geq 2$.

By Definition 4.1.4, there exists an annotated trace t' such that $P \xrightarrow{t'} \epsilon$ and $t' \sim W'$, where

$$W' = \left(\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \left\langle Q'_{m-1}, r'_{m-1}, s'_{m-1} \right\rangle, \langle Q'_m, r', s'_m \rangle\right)$$

for some row constraint r'. By Figure 4-4, $t' = \text{Nil}$. By Definition 3.4.26, this case is not possible.

<u>Case 2:</u> 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 4-5, there exists a loop layout tree l' such that $l = (Q, |\sigma(Q)|) \searrow l'$ and $\sigma' \vdash P_1 \Downarrow_{\text{loops}} l'$.

<u>Case 2.1:</u> $W = \text{Nil}$.

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

$$t \in \textsc{GetAnnotatedTrace}(l).$$

By Definition 3.4.26, $t \sim W$.

<u>Case 2.2:</u> $W = \langle Q', r', s' \rangle$.

By Definition 3.4.20, $|\sigma(Q')| \simeq r'$. By Definition 4.3.1, $\mathcal{F}(P) = Q$. By Definition 4.1.4, $Q' \doteq_{\text{Nil,Nil,Nil}} Q$. By Definition 3.4.24, Definition 3.4.22, and Definition 3.4.4, $\sigma(Q) = \sigma(Q')$. 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 $t = \langle Q, |\sigma(Q)|, \text{NotLoop} \rangle \mathbin{@} t'$. By Algorithm 4,

$$t \in \textsc{GetAnnotatedTrace}(l).$$

By Definition 3.4.26, $t \sim W$.

<u>Case 2.3:</u> $W = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$, where $m \geq 2$.

By Definition 4.1.4, there exists an annotated trace t' such that $P \xrightarrow{t'} \epsilon$ and $t' \sim W'$, where

$$W' = \left(\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \left\langle Q'_{m-1}, r'_{m-1}, s'_{m-1} \right\rangle, \langle Q'_m, r', s'_m \rangle\right)$$

for some row constraint r'.

Let $t' = \langle Q'', r'', \lambda'' \rangle \mathbin{@} t''$. By Definition 3.4.26, $Q'' \doteq_{\text{Nil,Nil,Nil}} Q'_1$. Also, $r'' \simeq r'_1$. By Figure 4-4, $Q'' \doteq_{\text{Nil,Nil,Nil}} Q$.

Hence $Q'_1 \doteq_{\text{Nil,Nil,Nil}} Q$. By Definition 3.4.24, Definition 3.4.22, and Definition 3.4.4, $\sigma(Q) = \sigma(Q'_1)$.

Since σ satisfies W, by Definition 3.4.20, 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 4-4, $P_1 \xrightarrow{t''} \epsilon$.

Let $W'' = \left(\langle Q'_2, r'_2, s'_2 \rangle, \ldots, \left\langle Q'_{m-1}, r'_{m-1}, s'_{m-1} \right\rangle, \langle Q'_m, r', s'_m \rangle\right)$. By

Definition 3.4.26, $t'' \sim W''$.

Let $W''' = (\langle Q'_2, r'_2, s'_2 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$. By Definition 4.1.4, W''' is derived from P_1.

By Definition 3.4.20, 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 $\sigma' \vdash P_1 \Downarrow_{\text{loops}} l'$, by the induction hypothesis, there exists an annotated trace $t''' \in \textsc{GetAnnotatedTrace}(l')$ such that $t''' \sim W'''$.

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

Since $|\sigma(Q)| \simeq r'_1$, by Definition 3.4.26, $t \sim W$.

<u>Case 3:</u> 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.

<u>Case 3.1:</u> $|\sigma(Q)| \geq 1$.

Let $\sigma' = \sigma[Q.y \mapsto \sigma(Q)]$. By Figure 4-5, there exists a loop layout tree l' such that $l = (Q, |\sigma(Q)|) \searrow l'$ and $\sigma' \vdash P_1 \Downarrow_{\text{loops}} l'$.

<u>Case 3.1.1:</u> $W = \text{Nil}$. The proof is similar to the proof of Case 2.1.

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

<u>Case 3.1.3:</u> $W = (\langle Q'_1, r'_1, s'_1 \rangle, \ldots, \langle Q'_m, r'_m, s'_m \rangle)$, 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 3.4.19, $r'_1 = (\geq 1)$ or $r'_1 = (\geq 2)$. Either case, since $r'' \simeq r'_1$, we have $r'' \geq 1$. Since $P \xrightarrow{t'} \epsilon$, by Figure 4-4, $P_1 \xrightarrow{t''} \epsilon$.

<u>Case 3.2:</u> $|\sigma(Q)| = 0$. 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 4-3, 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 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 4.1.4, $s'_1 = \texttt{true}$. By Definition 3.4.20, 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 $l = (Q, r) \circlearrowright (l'_1, \ldots, l'_r)$, by Algorithm 4, $t \in \textsc{GetAnnotatedTrace}(l)$.

Since $r = |\sigma(Q)| \simeq r'_1$, by Definition 3.4.26, $t \sim W$.

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 3.4.26 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 3.4.26 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 4-1, Algorithm 8, and Lemma 4.2.14. This prefix of the path constraint is useful for reasoning about the equivalence of the queries in t'' and W''.

□

Lemma 4.5.8. *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. <u>Case 1:</u> $W = \text{Nil}$.

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 3.4.26, $t \sim \text{Nil}$. In Algorithm 4, execution enters the branch on line 6 with $t \neq \text{Nil}$.

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

By Lemma 4.5.7, there exists an annotated trace

$$t' \in \textsc{GetAnnotatedTrace}(l)$$

such that $t' \sim W$. Since $m \geq 1$, by Definition 3.4.26, $t' \neq \text{Nil}$. In Algorithm 4, execution eventually enters the branch on line 6 with variable $t \neq \text{Nil}$.

□

Lemma 4.5.9. *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. *$l \hookrightarrow^{t} Nil$.*

Proof. <u>Case 1:</u> $l = \text{Nil}$.

By Figure 4-3, $P = \epsilon$. By Algorithm 4, $\textsc{GetAnnotatedTrace}(l) = \{\,\text{Nil}\,\}$. In Algorithm 4, execution never enters line 6 and thus returns Nil on line 10. Hence $t = \textsc{MatchPath}(l, W) = \text{Nil}$. By Figure 4-4, $P \xrightarrow{t} \epsilon$. By Figure 4-5, $l \hookrightarrow^{t} \text{Nil}$.

By Definition 4.1.4, $W = \text{Nil}$. By Definition 3.4.26, $t \sim W$.

<u>Case 2:</u> $l \neq \text{Nil}$.

By Lemma 4.5.8, $t = \textsc{MatchPath}(l, W) \neq \text{Nil}$. In Algorithm 4, execution must not return on line 10. Since $\textsc{GetAnnotatedTrace}(l)$ contains a finite number of annotated traces, the execution must return on line 7. Hence $t \in \textsc{GetAnnotatedTrace}(l)$ and $t \sim W$.

By Proposition 4.5.3, $P \xrightarrow{t} \epsilon$. By Proposition 4.5.6, $l \hookrightarrow^{t} \text{Nil}$.

□

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

1. $\sigma \vdash P \Downarrow_{\text{loops}} l'$,
2. $t \sim W$,

3. $P \xrightarrow{t} \epsilon$,

4. $l' \hookrightarrow^{t}$ 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 4.1.7, the variable e in Algorithm 2 and e' are identical except for equivalent variables.

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

Let $t' = \textsc{MatchPath}(l', W)$. Since the variable $t = \textsc{MatchPath}(l, W)$, t and t' are also identical except for equivalent variables. By Lemma 4.5.9, $t' \sim W$, $P \xrightarrow{t'} \epsilon$, and $l' \hookrightarrow^{t'}$ Nil.

By Figure 4-4, $P \xrightarrow{t} \epsilon$. By Definition 3.4.26, $t \sim W$. By Figure 4-5, $l' \hookrightarrow^{t}$ Nil.

□

4.6 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 4.6.1). We then present the soundness proof of InferProg in Section 4.6.2. We conclude with the soundness proof of Infer in Section 4.6.3.

4.6.1 Updating the Context while Traversing the Program AST

Figure 4-6 presents the definition of simultaneously updating the context, traversing a program $P \in$ Prog, and traversing a loop layout tree, by following an annotated trace.

Proposition 4.6.1. For any programs $P_1, P_2, P_3 \in$ Prog, contexts $\sigma_1, \sigma_2, \sigma_3 \in \mathit{Context}$, loop layout trees l_1, l_2, l_3, and annotated traces t_1, t_2:

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

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 $\begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix} \xrightarrow{t_2}\!\!\!\!\rightarrow \begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}$ and $\begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix} \xrightarrow{t_1 \,@\, t_2}\!\!\!\!\rightarrow \begin{bmatrix}\sigma_3\\P_3\\l_3\end{bmatrix}$, $\begin{bmatrix}\sigma_1\\P_1\\l_1\end{bmatrix} \xrightarrow{t_1}\!\!\!\!\rightarrow \begin{bmatrix}\sigma_2\\P_2\\l_2\end{bmatrix}$ may not hold. Counter examples are similar to that of Sections 4.5.1 and 4.5.2.

Proposition 4.6.2. For any programs $P, P' \in$ Prog, contexts $\sigma, \sigma' \in$ *Context*, loop layout trees l, l', annotated trace t, if $\begin{bmatrix}\sigma\\P\\l\end{bmatrix} \xrightarrow{t}\!\!\!\!\rightarrow \begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}$ then $P \xrightarrow{t} P'$ and $l \overset{t}{\hookrightarrow} l'$.

Proof. By induction on the derivation of P. □

Proposition 4.6.3. 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 $\sigma \vdash P \Downarrow_{\text{loops}} l$, $P \xrightarrow{\langle Q',r,\lambda\rangle} P'$, and $l \overset{\langle Q',r,\lambda\rangle}{\hookrightarrow} l'$, then there exists $\sigma' \in$ *Context* such that $\begin{bmatrix}\sigma\\P\\l\end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle}\!\!\!\!\rightarrow \begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}$.
2. for any context $\sigma' \in$ *Context*, if $\sigma \vdash P \Downarrow_{\text{loops}} l$ and $\begin{bmatrix}\sigma\\P\\l\end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle}\!\!\!\!\rightarrow \begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}$ then $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.

Proof. 1. By induction on the derivation of P.

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

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

By Figure 4-6, it is not possible to have $\begin{bmatrix}\sigma\\P\\l\end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle}\!\!\!\!\rightarrow \begin{bmatrix}\sigma'\\P'\\l'\end{bmatrix}$.

<u>Case 2:</u> 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 4-6, $r = |\sigma(Q)|$, $\sigma' = \sigma[Q.y \mapsto \sigma(Q)]$, $P' = P_1$, and $l = (Q, r) \searrow l'$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Figure 4-3, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.

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 4-6, $r = |\sigma(Q)| \geq 2$, $\lambda \in \{1, \ldots, r\}$, $\sigma' = \sigma[Q.y \mapsto x_\lambda]$, $P' = P_1$, 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 4-3, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.

□

Proposition 4.6.4. For any programs $P, P' \in$ Prog, context $\sigma \in$ *Context*, loop layout trees l, l', and annotated trace t:

1. if $\sigma \vdash P \Downarrow_{\text{loops}} l$, $P \xrightarrow{t} P'$, and $l \overset{t}{\hookrightarrow} l'$, then there exists $\sigma' \in$ *Context* such that $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \overset{t}{\twoheadrightarrow} \begin{bmatrix} \sigma' \\ P' \\ l' \end{bmatrix}$.
2. for any context $\sigma' \in$ *Context*, if $\sigma \vdash P \Downarrow_{\text{loops}} l$ and $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \overset{t}{\twoheadrightarrow} \begin{bmatrix} \sigma' \\ P' \\ l' \end{bmatrix}$ then $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.

Proof. 1. This proof is by induction on the length of t.

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

By Figure 4-4, $P' = P$. By Figure 4-5, $l' = l$. By Figure 4-6, $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \overset{\text{Nil}}{\twoheadrightarrow} \begin{bmatrix} \sigma \\ P \\ l \end{bmatrix}$.

Case 2: $t = \langle Q', r, \lambda \rangle \mathbin{@} t''$.

By Proposition 4.5.2, there exists $P'' \in \text{Prog}$ such that $P \xrightarrow{\langle Q',r,\lambda\rangle} P''$. By Proposition 4.5.5, there exists l'' such that $l \xhookrightarrow{\langle Q',r,\lambda\rangle} l''$. By Proposition 4.6.3, there exists $\sigma'' \in \mathit{Context}$ such that $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle}\!\!\!\!\rightarrow \begin{bmatrix} \sigma'' \\ P'' \\ l'' \end{bmatrix}$. By Proposition 4.6.3, $\sigma'' \vdash P'' \Downarrow_{\text{loops}} l''$.

Since $P \xrightarrow{t} P'$, we have $P \xrightarrow{\langle Q',r,\lambda\rangle \,@\, t''} P'$. By Proposition 4.5.1, $P'' \xrightarrow{t''} P'$.

Since $l \xhookrightarrow{t} l'$, we have $l \xhookrightarrow{\langle Q',r,\lambda\rangle \,@\, t''} l'$. By Proposition 4.5.4, $l'' \xhookrightarrow{t''} l'$.

By the induction hypothesis, there exists $\sigma''' \in \mathit{Context}$ such that $\begin{bmatrix} \sigma'' \\ P'' \\ l'' \end{bmatrix} \xrightarrow{t''}\!\!\!\!\rightarrow \begin{bmatrix} \sigma''' \\ P' \\ l' \end{bmatrix}$. Since $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle}\!\!\!\!\rightarrow \begin{bmatrix} \sigma'' \\ P'' \\ l'' \end{bmatrix}$, by Proposition 4.6.1,

$$\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle \,@\, t''}\!\!\!\!\rightarrow \begin{bmatrix} \sigma''' \\ P' \\ l' \end{bmatrix}.$$

Hence $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \xrightarrow{t}\!\!\!\!\rightarrow \begin{bmatrix} \sigma''' \\ P' \\ l' \end{bmatrix}$.

2. This proof is by induction on the length of t.

<u>Case 1:</u> $t = \text{Nil}$.

By Figure 4-6, $\sigma' = \sigma$, $P' = P$, 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 4.6.2, $P \xrightarrow{t} P'$ and $l \xhookrightarrow{t} l'$. By Proposition 4.5.2, there exists $P'' \in \text{Prog}$ such that $P \xrightarrow{\langle Q',r,\lambda\rangle} P''$. By Proposition 4.5.5, there exists l'' such that $l \xhookrightarrow{\langle Q',r,\lambda\rangle} l''$.

Since $\sigma \vdash P \Downarrow_{\text{loops}} l$, by Proposition 4.6.3, there exists σ'' such that $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \xrightarrow{\langle Q',r,\lambda\rangle}\!\!\!\!\rightarrow \begin{bmatrix} \sigma'' \\ P'' \\ l'' \end{bmatrix}$. By Proposition 4.6.3, $\sigma'' \vdash P'' \Downarrow_{\text{loops}} l''$.

Since $\begin{bmatrix} \sigma \\ P \\ l \end{bmatrix} \xrightarrow{t}\!\!\!\!\rightarrow \begin{bmatrix} \sigma' \\ P' \\ l' \end{bmatrix}$, by Proposition 4.6.1, $\begin{bmatrix} \sigma'' \\ P'' \\ l'' \end{bmatrix} \xrightarrow{t''}\!\!\!\!\rightarrow \begin{bmatrix} \sigma' \\ P' \\ l' \end{bmatrix}$.

By the induction hypothesis, $\sigma' \vdash P' \Downarrow_{\text{loops}} l'$.

$\square$

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

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

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

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

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

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

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

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

Figure 4-6: Traverse a program and a corresponding loop layout tree by following an annotated trace, updating the context

4.6.2 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 4.6.5. Consider any programs $P \in \text{Prog}$ and $P' \in \mathcal{K}$ and annotated traces t, t' such that $t \neq \text{Nil}$, $t' \neq \text{Nil}$, $P' \xrightarrow{t'} P$, 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 4.5.1, $P' \xrightarrow{t' @ t} \epsilon$. By Definition 3.4.26, Definition 4.1.4, and the definition of MakePathConstraint, W_i is derived from P'. □

Lemma 4.6.6. *Consider any programs $P \in \text{Prog}$ and $P' \in \mathcal{K}$ and annotated traces t, t' such that $t \neq \text{Nil}$, $t' \neq \text{Nil}$, $P' \xrightarrow{t'} P$, 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 $\begin{bmatrix} \sigma'_i \\ P' \\ l'_i \end{bmatrix} \xrightarrow{\langle Q_1, r_{i,1}, \lambda_{i,1} \rangle, \ldots, \langle Q_k, r_{i,k}, \lambda_{i,k} \rangle} \begin{bmatrix} \sigma_i \\ P \\ l_i \end{bmatrix}$ and $\begin{bmatrix} \sigma_i \\ P \\ l_i \end{bmatrix} \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle} \begin{bmatrix} \sigma''_i \\ P'' \\ l''_i \end{bmatrix}$.*

Proof. In Algorithm 10, variables $s_1 = t'$ and $s_2 = t$. By Proposition 4.6.5, 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:

$$\sigma'_i \vdash P' \Downarrow_{\text{loops}} l'_i \tag{4.1}$$

$$t_i \sim W_i \tag{4.2}$$

$$P' \xrightarrow{t_i} \epsilon \tag{4.3}$$

$$l'_i \hookrightarrow \text{Nil} \tag{4.4}$$

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

$$t_i = t_{i,1} \,@\, t_{i,2} \tag{4.5}$$

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:

$$t_{i,1} = t'_{i,1} \,@\, \langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle \tag{4.6}$$

By (4.4), (4.5), (4.6), and Proposition 4.5.5, there exists l_i, l''_i such that:

$$l'_i \xhookrightarrow{t'_{i,1}} l_i \tag{4.7}$$

$$l_i \xhookrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle} l''_i \tag{4.8}$$

For all $j = 1, \ldots, k$, by (4.2), 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 4-4. Since $P' \xrightarrow{t'} P$,

$$P' \xrightarrow{t'_{i,1}} P \tag{4.9}$$

By (4.1), (4.9), (4.7), and Proposition 4.6.4, there exists σ_i such that:

$$\begin{bmatrix} \sigma'_i \\ P' \\ l'_i \end{bmatrix} \xrightarrow{t'_{i,1}}\!\!\!\!\rightarrow \begin{bmatrix} \sigma_i \\ P \\ l_i \end{bmatrix} \tag{4.10}$$

By (4.3), (4.5), (4.6), (4.9), and Proposition 4.5.1,

$$P \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle \,@\, t_{i,2}} \epsilon \tag{4.11}$$

By (4.11) and Proposition 4.5.2, there exists P'' such that:

$$P \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle} P'' \tag{4.12}$$

By (4.1), (4.10), and Proposition 4.6.4,

$$\sigma_i \vdash P \Downarrow_{\text{loops}} l_i \tag{4.13}$$

By (4.13), (4.12), (4.8), and Proposition 4.6.4, there exists σ_i'' such that

$$\begin{bmatrix} \sigma_i \\ P \\ l_i \end{bmatrix} \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle} \!\!\!\!\rightarrow \begin{bmatrix} \sigma_i'' \\ P'' \\ l_i'' \end{bmatrix}.$$

□

Lemma 4.6.7. *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*

$$\textsc{InferProg}(\boxed{P'}, t', t),$$

if variable $f_2 =$ `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 4.6.5, W_2 is derived from P'.

Let σ_2' be the context variable in SolveAndGetTrace on line 10 for $i = 2$. Since $f_2 =$ `true`, by Algorithm 5, variable $t_2 = \textsc{GetTrace}(\boxed{P'}, W_2, \sigma_2')$. 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 $l_2' \xhookrightarrow{t_2}$ Nil.

<u>Case 1:</u> $t' =$ Nil.

Since $P' \xrightarrow{t'} P$, by Figure 4-4, $P' = P$.

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

Since $t' =$ 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 $t_2 = t_{2,1} \mathbin{@} t_{2,2} = \langle Q_1, r_{2,1}, \lambda_{2,1} \rangle \mathbin{@} t_{2,2}$.

Since $l_2' \xhookrightarrow{t_2}$ Nil, by Proposition 4.5.5, 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 3.4.26, 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 4-4, there exists P'' such that $P \xrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1} \rangle} P''$.

By Proposition 4.6.3, there exists σ'' such that $\begin{bmatrix} \sigma_2 \\ P \\ l_2 \end{bmatrix} \xrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1} \rangle} \begin{bmatrix} \sigma_2'' \\ P'' \\ l_2'' \end{bmatrix}$.

<u>Case 2:</u> $t' \neq \text{Nil}$.

By Lemma 4.6.6, there exists $\sigma_2, \sigma_2'', P'', l_2, l_2''$ such that

$$\begin{bmatrix} \sigma_2' \\ P' \\ l_2' \end{bmatrix} \xrightarrow{\langle Q_1, r_{2,1}, \lambda_{2,1} \rangle, \ldots, \langle Q_k, r_{2,k}, \lambda_{2,k} \rangle} \begin{bmatrix} \sigma_2 \\ P \\ l_2 \end{bmatrix}$$

and

$$\begin{bmatrix} \sigma_2 \\ P \\ l_2 \end{bmatrix} \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1} \rangle} \begin{bmatrix} \sigma_2'' \\ P'' \\ l_2'' \end{bmatrix}.$$

Either case, we have $\begin{bmatrix} \sigma_2 \\ P \\ l_2 \end{bmatrix} \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1} \rangle} \begin{bmatrix} \sigma_2'' \\ P'' \\ l_2'' \end{bmatrix}$.

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

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

Since $P \xrightarrow{t} \epsilon$, by Figure 4-4, it is not possible to have $t \neq \text{Nil}$. Hence the proposition trivially holds.

<u>Case 2:</u> 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 $\begin{bmatrix} \sigma_2 \\ P \\ l_2 \end{bmatrix} \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1} \rangle} \begin{bmatrix} \sigma_2'' \\ P'' \\ l_2'' \end{bmatrix}$, by Figure 4-6, $\lambda_{2,k+1} = \text{NotLoop}$.

<u>Case 3:</u> 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 3.4.26 and the definition of MakePathConstraint, $r_{2,k+1} \geq 2$. Since $\begin{bmatrix} \sigma_2 \\ P \\ l_2 \end{bmatrix} \xrightarrow{\langle Q_{k+1}, r_{2,k+1}, \lambda_{2,k+1} \rangle} \!\!\!\!\rightarrow \begin{bmatrix} \sigma_2'' \\ P'' \\ l_2'' \end{bmatrix}$, by Figure 4-6, $\lambda_{2,k+1} \neq \text{NotLoop}$.

□

Lemma 4.6.8. *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 4.6.5, W_i is derived from P'.

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

$$t_i \sim W_i \tag{4.14}$$

$$P' \xrightarrow{t_i} \epsilon \tag{4.15}$$

Since $P \xrightarrow{t} \epsilon$, by Figure 4-4, $Q \doteq Q_{k+1}$. Hence, by Figure 4-4,

$$P \xrightarrow{\langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle} P_1 \tag{4.16}$$

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 (4.16), $P \xrightarrow{t_{i,1}} P_1$.

Since $P' \xrightarrow{t'} P$, by Figure 4-4, $P' = P$. By (4.15), $P \xrightarrow{t_i} \epsilon$.

Since $t_i = t_{i,1} @ t_{i,2}$, by Proposition 4.5.1, $P_1 \xrightarrow{t_{i,2}} \epsilon$.

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:

$$t_i = t_{i,1} \,@\, t_{i,2} \tag{4.17}$$

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:

$$t_{i,1} = t'_{i,1} \,@\, \langle Q_{k+1}, r_{i,k+1}, \lambda_{i,k+1} \rangle \tag{4.18}$$

By (4.14), Definition 3.4.26, and the definition of 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 4-4. Since $P' \xrightarrow{t'} P$,

$$P' \xrightarrow{t'_{i,1}} P \tag{4.19}$$

By (4.19), (4.16), (4.18), and Proposition 4.5.1,

$$P' \xrightarrow{t_{i,1}} P_1 \tag{4.20}$$

By (4.15), (4.20), (4.17), and Proposition 4.5.1,

$$P_1 \xrightarrow{t_{i,2}} \epsilon \tag{4.21}$$

□

Lemma 4.6.9. *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* InferProg($\boxed{P'}, t', t$)*, if variables $f_0 = f_1 =$ `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 $\pi_{\mathcal{S}} Q_{0,k+2} = \pi_{\mathcal{S}} Q_{1,k+2}$.*

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 4.6.8, $P_1 \xrightarrow{t_{i,2}} \epsilon$ 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 4-4, $t_{i,2} = \text{Nil}$ 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 4-4, for each $i = 0, 1$ we have $t_{i,2} \neq \text{Nil}$ and $Q_{i,k+2} \doteq Q'$. Hence $\pi_{\mathcal{S}} Q_{0,k+2} = \pi_{\mathcal{S}} Q_{1,k+2} = \pi_{\mathcal{S}} Q'$.

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

Proof. 1. By Proposition 4.6.5, 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 4.6.8.

2. The proof is similar to the proof of 1.

□

Lemma 4.6.11. *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 $\textsc{InferProg}(\boxed{P'}, t', t)$*:*

1. *if variable $f_0 =$ `true` on line 16 then $P' \xrightarrow{t_{0,1}} P_2$ and $P_2 \xrightarrow{t_{0,2}} \epsilon$.*

2. *if variable $f_2 =$ `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 4.6.10. □

Theorem 3 (Core Recursion)**.** For any programs $P \in \text{Prog}$ 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. This proof is by induction on the derivation of P.

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

By Figure 4-4, $t = \text{Nil}$. By Algorithm 10, $\textsc{InferProg}(\boxed{P'}, t', \text{Nil}) = \epsilon$.

<u>Case 2:</u> 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 4.6.7, if $f_2 = \texttt{true}$ then $\lambda_{2,k+1} = \text{NotLoop}$, so execution does not enter the branch on line 17. By Lemma 4.6.9, execution does not enter the branch on line 23. Hence, execution enters the branch on line 27.

Since $P' \xrightarrow{t'} P$, by Figure 4-4, P is a subprogram of P'. Hence Q is a query in P'. Since $P' \in \mathcal{K}$, by Definition 3.3.4 and Proposition 4.2.6, $\textsc{Trim}(P') = P'$. Hence Q is a query in $\textsc{Trim}(P')$. By Proposition 4.2.17, there exists a context $\sigma \in \mathit{Context}$ 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 4.1.6, at least one of the variables f_0, f_1 is `true`.

If $f_i = \texttt{true}$ $(i = 0, 1)$, by Lemma 4.6.8, $P' \xrightarrow{t_{i,1}} P_1$ and $P_1 \xrightarrow{t_{i,2}} \epsilon$. By the induction hypothesis, $P_1 \doteq \textsc{InferProg}(\boxed{P'}, t_{i,1}, t_{i,2})$. Either case, P_1 and variable b on line 31 are identical except for equivalent variables.

Since $P \xrightarrow{t} \epsilon$, by Figure 4-4, $Q \doteq Q_{k+1}$.

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 4.6.7, if $f_2 = \texttt{true}$ then $\lambda_{2,k+1} = \text{NotLoop}$, so execution does not enter the branch on line 17.

Since $P' \xrightarrow{t'} P$, by Figure 4-4, P is a subprogram of P'. Hence Q is a query in P'. Since $P' \in \mathcal{K}$, by Definition 3.3.4 and Proposition 4.2.6, $\textsc{Trim}(P') = P'$. Hence Q is a query in $\textsc{Trim}(P')$. By Proposition 4.2.17, there exists a context $\sigma \in \mathit{Context}$ such that Q is used while evaluating $\sigma(P')$ and the corresonding row count is zero (or positive). So both of the path constraints W_0, W_1 are satisfiable. By Proposition 4.1.6, variables $f_0 = f_1 = \texttt{true}$.

Since $\textsc{Trim}(P') = P'$ and P is subprogram of P', by Algorithm 8, it is not possible to have $P_1 = P_2 = \epsilon$. By Definition 3.3.4 and Definition 4.3.6, $\pi_{\mathcal{S}}\mathcal{F}(P_1) \neq \pi_{\mathcal{S}}\mathcal{F}(P_2)$.

By Lemma 4.6.10, $P' \xrightarrow{t_{0,1}} P_2$ and $P_2 \xrightarrow{t_{0,2}} \epsilon$. By Lemma 4.6.10, $P' \xrightarrow{t_{1,1}} P_1$ and $P_1 \xrightarrow{t_{1,2}} \epsilon$.

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

By Figure 4-4, $t_{1,2} = \text{Nil}$ and $t_{0,2} \neq \text{Nil}$.

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

By Figure 4-4, $t_{1,2} \neq \text{Nil}$ and $t_{0,2} = \text{Nil}$.

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

By Figure 4-4, $t_{1,2} \neq \text{Nil}$ and $t_{0,2} \neq \text{Nil}$. $Q_{0,k+2} \doteq \mathcal{F}(P_2)$. $Q_{1,k+2} \doteq \mathcal{F}(P_1)$. 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

$$b_f = \textsc{InferProg}(\boxed{P'}, t_{0,1}, t_{0,2})$$

on line 26 are identical except for equivalent variables. Also, P_1 and the variable $b_t = \textsc{InferProg}(\boxed{P'}, t_{1,1}, t_{1,2})$ are identical except for equivalent variables.

Since $P \xrightarrow{t} \epsilon$, by Figure 4-4, $Q \doteq Q_{k+1}$.

<u>Case 4:</u> 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 4-4, P is a subprogram of P'. Hence Q is a query in P'. Since $P' \in \mathcal{K}$, by Definition 3.3.4 and Proposition 4.2.6, $\textsc{Trim}(P') = P'$. Hence Q is a query in $\textsc{Trim}(P')$. By Proposition 4.2.17, there exists a context $\sigma \in \mathit{Context}$ such that Q is used while evaluating $\sigma(P')$ and the corresonding row count is at least two. So the path constraint W_2 is satisfiable. By Proposition 4.1.6, variable $f_2 = \texttt{true}$.

By Lemma 4.6.7, variable $\lambda_{2,k+1} \neq$ NotLoop. Execution enters the branch on line 17.

By Lemma 4.6.11, $P' \xrightarrow{t_{2,1}} P_1$ and $P_1 \xrightarrow{t_{2,2}} \epsilon$. By the induction hypothesis, P_1 and the variable $b_t = \textsc{InferProg}(\boxed{P'}, t_{2,1}, t_{2,2})$ on line 22 are identical except for equivalent variables.

<u>Case 4.1:</u> $f_0 = \texttt{true}$.

By Lemma 4.6.11, $P' \xrightarrow{t_{0,1}} P_2$ 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.

<u>Case 4.2:</u> $f_0 = \texttt{false}$.

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 4-4, $Q \doteq Q_{k+1}$.

□

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(P, s_1, s_2)
    if s_2 = Nil then return ϵ                                   ▷ Prog := ϵ
    end if
    k ← The length of s_1
    if k > 0 then ⟨Q_1, r_1, λ_1⟩, ..., ⟨Q_k, r_k, λ_k⟩ ← s_1
    end if
    ⟨Q_{k+1}, r_{k+1}, λ_{k+1}⟩, ..., ⟨Q_n, r_n, λ_n⟩ ← s_2
    for i = 0, 1, 2 do
        W_i ← MakePathConstraint(s_1, Q_{k+1}, i)
        (f_i, t_i) ← SolveAndGetTrace(P, W_i)
        if f_i then                                              ▷ Satisfiable
            ⟨Q_1, r_{i,1}, λ_{i,1}⟩, ..., ⟨Q_{k+1}, r_{i,k+1}, λ_{i,k+1}⟩,
                ⟨Q_{i,k+2}, r_{i,k+2}, λ_{i,k+2}⟩, ..., ⟨Q_{i,m_i}, r_{i,m_i}, λ_{i,m_i}⟩ ← t_i
            t_{i,1} ← ⟨Q_1, r_{i,1}, λ_{i,1}⟩, ..., ⟨Q_{k+1}, r_{i,k+1}, λ_{i,k+1}⟩   ▷ New trace prefix
            t_{i,2} ← ⟨Q_{i,k+2}, r_{i,k+2}, λ_{i,k+2}⟩, ..., ⟨Q_{i,m_i}, r_{i,m_i}, λ_{i,m_i}⟩   ▷ New trace suffix
        end if
    end for
    if f_2 and λ_{2,k+1} ≠ NotLoop 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_{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} ≠ Nil) or (t_{0,2} ≠ Nil and t_{1,2} = Nil) or
                (t_{0,2} ≠ Nil and t_{1,2} ≠ Nil and π_S Q_{0,k+2} ≠ π_S Q_{1,k+2})) then
        b_t ← InferProg(P, t_{1,1}, t_{1,2})
        b_f ← InferProg(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 ← InferProg(P, t_{0,1}, t_{0,2})
        else b ← InferProg(P, t_{1,1}, t_{1,2})
        end if
        return "Q_{k+1} b"                                       ▷ Prog := Seq
    end if
end procedure
```

4.6.3 Soundness of Infer

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

Proof. By Definition 3.4.20, the variable σ in Algorithm 1 satisfies the trivial path constraint Nil. By Figure 4-4, there exists an annotated trace t' such that $P \xrightarrow{t'} \epsilon$. By Definition 3.4.26, $t' \sim \text{Nil}$. By Definition 4.1.4, 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 4-4, $P \xrightarrow{\text{Nil}} P$. By Theorem 3, $P \doteq \textsc{InferProg}(\boxed{P}, \text{Nil}, t)$. □

Corollary 4.6.12. *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 4.2.8, $\textsc{Infer}(\boxed{P_1}) = \textsc{Infer}(\boxed{P_2})$. 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 4.6.13. *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 4.2.13 and Corollary 4.6.12. □

4.7 Complexity

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

Lemma 4.7.1. *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.

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

By Figure 4-4, $t = \text{Nil}$. By Algorithm 6, the procedure returns immediately without calling InferProg. By Definition 4.1.5, $\|\epsilon\| = 1$.

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 `true`. When f_i = `true` ($i = 0, 1$), by the induction hypothesis, 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 4.1.5, $\|P\| = 1 + \|P_1\|$.

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 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 4.1.5, $\|P\| = 1 + \|P_1\| + \|P_2\|$.

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 4.7.1, 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. □

4.8 Remark on the Konure DSL

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

4.8.1 Programs in Konure DSL Grammar

Apart from the set of inferrable programs $\mathcal{K}$ (Definition 3.3.4) for which we designed Konure, we also identify the following interesting sets of programs in Prog, where we obtain a stronger result.

Definition 4.8.1.

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

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 4.8.2. *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 3.3.2 and Theorem 7, $\widetilde{P_1} \equiv P_1$ and $\widetilde{P_2} \equiv P_2$. By Definition 4.2.7, $\widetilde{P_1} \equiv \widetilde{P_2}$. By the definition of K_2, $\widetilde{P_1}, \widetilde{P_2} \in \mathcal{K}$. By Corollary 4.6.12, $\widetilde{P_1} \doteq \widetilde{P_2}$. □

Corollary 4.8.3. *For any program $P \in K_3$, let $P' \in \mathcal{K}$ such that $P \equiv P'$, then* $P' \doteq \textsc{Infer}(\boxed{P})$.

Proof. By the definition of K_3, such program P' exists. Since $P \equiv P'$, by Proposition 4.2.8, $\textsc{Infer}(\boxed{P}) = \textsc{Infer}(\boxed{P'})$. By Theorem 4, $P' \doteq \textsc{Infer}(\boxed{P'})$. □

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

Proposition 4.8.4. $\mathcal{K} \subset K_2$.

Proof.

1. $\mathcal{K} \subseteq K_2$: For any program $P \in \mathcal{K}$, by Definition 3.3.4, there exists program $P' \in \text{Prog}$ such that $P = \widetilde{P'}$. By Definition 3.3.2 and Proposition 4.2.6, $\widetilde{\widetilde{P'}} = \widetilde{P'}$. 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 $Q_1, Q_2 \in \text{Query}$ such that $\pi_{\mathsf{S}} Q_1 \neq \pi_{\mathsf{S}} Q_2$ 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 $P \in \text{Prog}$ 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 3.3.2,

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

By Definition 3.3.4, $\widetilde{P} \in \mathcal{K}$. By the definition of K_2, $P \in K_2$. Since $P \neq \widetilde{P}$, by Proposition 4.2.6, there does not exist any program $P' \in \text{Prog}$ such that $P = \widetilde{P'}$. By Definition 3.3.4, $P \notin \mathcal{K}$.

□

Proposition 4.8.5. $K_2 \subset K_3$.

Proof.

1. $K_2 \subseteq K_3$: For any program $P \in K_2$, by definition, $\widetilde{P} \in \mathcal{K}$. By Definition 3.3.2 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 $\sigma, \sigma' \in$ *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 4.2.7, $P_1 \equiv P_2$. By Definition 3.3.2, $\widetilde{P_1} = P_1$ and $\widetilde{P_2} = P_2$. By Definition 3.3.4, $P_1 \in \mathcal{K}$ and $P_2 \notin \mathcal{K}$. Hence $P_2 \in K_3$ and $P_2 \notin K_2$.

□

Proposition 4.8.6. $K_3 \subset K_4$.

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 4.2.13, $P' \equiv \textsc{Infer}(\boxed{P'})$. By Definition 4.2.7, $\textsc{Infer}(\boxed{P'}) \equiv P$.

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 3.3.2, $\widetilde{P} = P$. 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 3.3.4, $\widetilde{P} \notin \mathcal{K}$. 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 3.6). 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 4.2.13, $P \equiv$ 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, $P' \doteq \textsc{Infer}(\boxed{P'})$. Since $P \equiv P'$, the black box programs $\boxed{P}$ and $\boxed{P'}$ are observationally equivalent. Hence $\textsc{Infer}(\boxed{P}) = \textsc{Infer}(\boxed{P'})$. Since $P \doteq \textsc{Infer}(\boxed{P})$, 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 4.8.7. $K_4 \subset$ 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 $\pi_{\mathcal{S}} Q_1$ and $\pi_{\mathcal{S}} Q_2$ 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 $\textsc{Infer}(\boxed{P})$ 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 4.8.4 states that the Trim transformation transforms certain programs that are not in the Konure DSL into equivalent programs in the Konure DSL. Proposition 4.8.5 states that the Trim transformation does not transform all of the potential programs into the Konure DSL. Proposition 4.8.6 states that the restrictions in Definition 3.3.4 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 4.8.7 states that the KONURE DSL syntax alone is not sufficient for inferrability.

4.8.2 Programs Expressible in KONURE DSL

Recall that two programs in Prog are observationally equivalent (Definition 4.2.7) if they produce the same concrete trace (Definition 3.4.6) for all contexts. In other words, when these programs are executed as black boxes (Definition 3.4.8), 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 4.8.8. $\textcircled{U}$ denotes the black box executable for a program with an unknown implementation. To execute $\textcircled{U}$ with a context $\sigma \in \mathit{Context}$, we populate the database, set the input parameters, and collect the concrete trace as in the EXECUTE procedure.

Definition 4.8.9. $\textcircled{U}$ is *expressible* as program $P \in \text{Prog}$ if for all contexts $\sigma \in \mathit{Context}$, executing $\textcircled{U}$ with σ produces $\sigma(P)$. $\textcircled{U}$ is expressible in $\mathcal{K}$ if there exists a program $P \in \mathcal{K}$ such that $\textcircled{U}$ is expressible as P.

Proposition 4.8.10. For any program $P \in K_3$, $\boxed{P}$ is expressible in $\mathcal{K}$.

Proof. By the definition of K_3 (Definition 4.8.1), there exists program $P' \in \mathcal{K}$ such that $P \equiv P'$. By Definition 4.2.7, for any context $\sigma \in \mathit{Context}$, $\sigma(P) = \sigma(P')$. By Definition 3.4.8, executing $\boxed{P}$ produces $\sigma(P')$. By Definition 4.8.9, $\boxed{P}$ is expressible as P' and is expressible in $\mathcal{K}$. □

Proposition 4.8.11. For any program $P \in \text{Prog}$, if $\textcircled{U}$ is expressible as P then $\textsc{Infer}(\textcircled{U}) = \textsc{Infer}(\boxed{P})$.

Proof. By Definition 4.8.9, for any context $\sigma \in \mathit{Context}$,

$$\textsc{Execute}(\textcircled{U}, \sigma) = \textsc{Execute}(\boxed{P}, \sigma).$$

By Algorithm 1, $\textsc{Infer}(\textcircled{U}) = \textsc{Infer}(\boxed{P})$. $\square$

Corollary 4.8.12. *For any program $P \in \mathcal{K}$, if $\textcircled{U}$ is expressible as P then $P \doteq \textsc{Infer}(\textcircled{U})$.*

Proof. By Proposition 4.8.11 and Theorem 4. $\square$

Corollary 4.8.12 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.

Chapter 5

Experimental Evaluation of Konure

We evaluate our program inference technique with the following research questions:

- **RQ1: Ability to Infer and Regenerate Benchmark Programs.** Can Konure infer real-world applications that access databases?

  We used Konure to infer the commands in several open-source database-backed applications (Section 5.1). We present positive results, where Konure infers the commands in reasonable amounts of time, in Section 5.1.2 and discuss limitations in Section 5.1.3.

- **RQ2: Scalability.** How does our technique scale with more complex programs?

  Our results indicate that Konure scales well for most dimensions of program complexity. The only dimension for which Konure does not scale well is the length of ambiguous long reference chains (Section 5.2).

- **RQ3: Active Learning Versus User Inputs.** How does our active learning technique compare with manual user inputs?

  Our results indicate that, in contrast to our active learning approach, the manual approach often misses infrequent corner cases (Section 5.3).

We performed experiments 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. Our KONURE implementation uses Python 3.5.3 (PyPy 6.0.0) and Z3 4.6.0.

5.1 RQ1: Ability to Infer and Regenerate Benchmark Programs

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.

5.1.1 Benchmark Applications

Our benchmark applications include:

- **Fulcrum Task Manager:** Fulcrum [3] is an open-source project planning tool, built with Ruby on Rails, with over 1500 stars on GitHub. The source code contains 3642 lines of JavaScript, Ruby, SASS, and HTML. Fulcrum maintains multiple projects. Each project may contain multiple stories. Each story may contain multiple notes. Fulcrum retrieves data from 5 relevant tables with 55 columns. Its 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 [5] is an open-source chat room application, built with Ruby on Rails, with over 2700 stars on GitHub. The source code contains 8438 lines of JavaScript, CoffeeScript, SASS, CSS, Ruby, and HTML. Kandan maintains multiple chat rooms (so-called channels) that users can access. Kandan retrieves data from 4 relevant tables with 41 columns. Its commands enable users to navigate chat rooms and messages (so-called activities) and display relevant user information.

- **Enki Blogging Application:** Enki [2] is an open-source blogging application, built with Ruby on Rails, with over 800 stars and 280 forks on GitHub. The source code contains 2589 lines of Ruby, HTML, JavaScript, CSS, and SASS. Enki maintains multiple pages and posts, each of which may have comments. Enki retrieves data from 5 relevant tables with 39 columns. Its 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 [4]. The source code contains 232 lines of HTML, Ruby, and JavaScript. Blog maintains information about blog articles and blog comments. Blog retrieves data from 2 relevant tables with 11 columns. 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 3.1. 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" [7], the version "TC_Java_89_m100" is the most similar to the program that we used and implements largely the same functionality. The application was written in Java and interacts with a MySQL database [165] via JDBC [123]. The source code contains 1264 lines of Java. It retrieves data from 5 relevant tables with 17 columns.

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. A copy of the source code for these benchmark applications is available at [10].

Application Selection Criteria. We choose our real world benchmark applications — Fulcrum, Kandan, and Enki — from the applications studied in a recent survey paper [173]. 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:

- 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.

- Some applications implement highly specialized calculations. For example, online shopping applications perform specific numeric calculations specific to that domain.

- 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 eight of 14 data retrieval commands as potential inference candidates. For Kandan, we identified six of 11, for Enki, four of ten, for Blog, two of two, and for Student Registration, one of one. 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 5.1.3.

5.1.2 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 Appendix A and at [6].

Table 5.1 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

Table 5.1: Inference effort and regenerated code size

Command	Params	App	Runs	Solves	Time	LoC	SQL	If	For	Output
`get_home`	1	Fulcrum	5	43	8 mins	21	5	1	0	9
`get_projects`	1	Fulcrum	5	43	8 mins	21	5	1	0	9
`get_projects_id`	2	Fulcrum	12	124	29 mins	25	8	2	0	8
`get_projects_id_stories`	2	Fulcrum	11	42	7 mins	31	8	3	0	11
`get_projects_id_stories_id`	3	Fulcrum	12	50	8 mins	31	9	3	0	11
`get_projects_id_stories_id_notes`	3	Fulcrum	11	41	8 mins	24	9	3	0	4
`get_projects_id_stories_id_notes_id`	4	Fulcrum	13	46	10 mins	28	10	4	0	4
`get_projects_id_users`	2	Fulcrum	12	124	30 mins	25	8	2	0	8
`get_channels`	1	Kandan	21	125	105 mins	63	16	4	2	27
`get_channels_id_activities`	2	Kandan	23	242	39 mins	49	16	6	0	13
`get_channels_id_activities_id`	3	Kandan	14	18	7 mins	25	11	3	0	3
`get_me`	1	Kandan	11	139	6 mins	44	8	3	0	25
`get_users`	1	Kandan	15	236	9 mins	67	11	3	0	45
`get_users_id`	2	Kandan	11	139	6 mins	44	8	3	0	25
`get_admin_comments_id`	1	Enki	2	5	22 secs	10	1	0	0	5
`get_admin_pages`	0	Enki	2	1	22 secs	13	2	1	0	4
`get_admin_pages_id`	1	Enki	2	5	23 secs	9	1	0	0	4
`get_admin_posts`	0	Enki	3	2	33 secs	16	3	1	1	3
`get_articles`	0	Blog	2	11	21 secs	12	2	0	0	6
`get_article_id`	1	Blog	6	29	42 secs	16	3	1	0	6
`liststudentcourses`	2	Student	6	20	41 secs	24	5	3	1	3

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. 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 **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 occasion-

ally 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) which 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.

5.1.3 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 3.3.4 and 3.3.5 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 val-

ues), KONURE is likely to omit the functionality for the rare corner cases and end up inferring only the main functionality.

Example 5.1.1. 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 5.1.2. 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 5.1.3. 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 5.1.4. 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 ($k = 1, 2, 3, \ldots$):

$$Q_1 = y_1 \leftarrow \texttt{select t1.id, t1.val where t1.id} = i\,;\ \texttt{print [t1.id]},$$
$$Q_{k+1} = y_{k+1} \leftarrow \texttt{select t1.id, t1.val where t1.id} = y_k.\texttt{t1.val}\,;\ \texttt{print [t1.id]}.$$

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.

5.2 RQ2: Scalability

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 B and at [6].

- **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.

5.2.1 Results

For each class above, we built representative commands with varying code sizes. We then used Konure to infer each command. Figure 5-1 presents statistics from running Konure on these synthetic commands. For SS commands (Figure 5-1a), the horizontal axis presents the number of queries in the command. For the remaining commands (Figures 5-1b-5-1e), 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** (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)

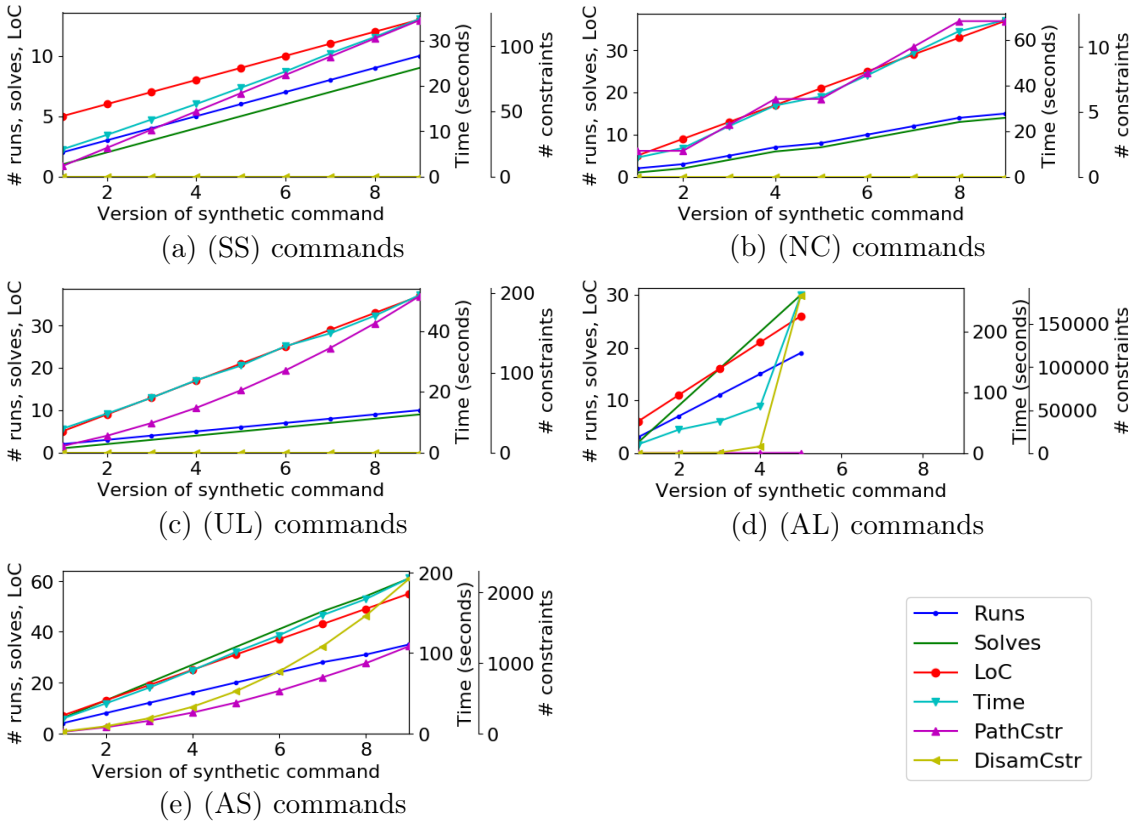


Figure 5-1: Performance on synthetic commands

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 5-1d, Konure ran out of memory after the version with five conditionals.

5.2.2 Discussion

Konure scales well for (SS), (NC), (UL), and (AS) commands, which is consistent with results in Section 5.1. Konure does not scale well for (AL) commands, where the major performance bottleneck is sending the solver disambiguation constraints (Section 3.6). 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 5.1 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.

5.3 RQ3: Active Learning Versus User Inputs

Instead of using active learning, an alternative approach is to use dynamic monitoring to obtain inputs for interacting with the seed program. 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 [134]. 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 [145] 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.

5.4 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.

Chapter 6

SHEAR: Inferring Loop Structures in Database-Backed Applications via Speculative Manipulation

In this chapter, we first present an example that illustrates how SHEAR works. We then present the design and implementation of SHEAR, including the problem formulation, the algorithm for inferring loop and repetitive structures, and the algorithm for inferring the full program structure.

6.1 Example

We present an example that illustrates how SHEAR uses speculative manipulation to analyze the loop and repetitive structures in a database program.

Example Program. Figure 6-1 presents the code of a task management program, where each function call `do_sql(query, params)` performs a database query by replacing the variables in `query` with the corresponding values specified in `params`. The program takes an input argument, `tid`, and retrieves data from three database tables: tasks, comments, and users. It first retrieves a task specified by the input. When the task exists, the program retrieves comments under this task. For each comment, the

```
tasks1 = do_sql("SELECT * FROM tasks WHERE id = :x",
                {"x": tid})
print(tasks1['title'])

if tasks1:
  comments = do_sql("SELECT * FROM comments WHERE task_id = :x",
                    {"x": tid})
  print(comments['content'])

  for c in comments:
    cid = c['commenter_id']
    users1 = do_sql("SELECT * FROM users WHERE id = :x",
                    {"x": cid})
    print(users1['name'])
    tasks2 = do_sql("SELECT * FROM tasks WHERE creator_id = :x",
                    {"x": cid})
    print(tasks2['title'])

  aid = tasks1['assignee_id']
  users2 = do_sql("SELECT * FROM users WHERE id = :x",
                  {"x": aid})
  print(users2['name'])
  tasks3 = do_sql("SELECT * FROM tasks WHERE creator_id = :x",
                  {"x": aid})
  print(tasks3['title'])
```

Figure 6-1: Example program in Python

program retrieves the user that made this comment, along with all tasks created by this user. After iterating over comments, the program retrieves the user to which the task is assigned, along with all tasks created by this user.

An example execution of the program uses the tasks and comments tables in Figure 6-2a, the users table empty, and the input `tid=2`. Figure 6-2b presents the resulting trace of the database traffic. This trace is an instance of the program behavior that we observe when executing the program.

Ambiguities from Loops and Repetitions. Without knowing the ground truth, our algorithm analyzes the trace to infer the program's loop structure. We first describe multiple plausible (but not equivalent) loop structures and then present how our algorithm disambiguates these plausible structures to infer the correct one.

We highlight five plausible loop structures in Figure 6-3. Each of these structures may produce the trace in Figure 6-2b. Each loop structure performs certain queries,

tasks:	id	title	creator_id	assignee_id
	1	1	4	6
	2	5	4	6

comments:	id	task_id	commenter_id	content
	3	2	4	7
	5	2	6	7

(a) Database tables

```
q0 : SELECT * FROM tasks WHERE id = 2
q1 : SELECT * FROM comments WHERE task_id = 2
q2 : SELECT * FROM users WHERE id = 4
q3 : SELECT * FROM tasks WHERE creator_id = 4
q4 : SELECT * FROM users WHERE id = 6
q5 : SELECT * FROM tasks WHERE creator_id = 6
q6 : SELECT * FROM users WHERE id = 6
q7 : SELECT * FROM tasks WHERE creator_id = 6
```

(b) SQL database traffic of an execution. The queries retrieve 1, 2, 0, 2, 0, 0, 0, and 0 rows, respectively.

Figure 6-2: Example execution trace

iterates over the rows retrieved by a query, and optionally performs more queries after the loop ends. We use comments (after the "#" symbol) to represent queries produced across different loop iterations. Among these candidate loop structures, the only one that is consistent with the program (Figure 6-1) is Plausible Loop L (Figure 6-3a). All other candidates are nonequivalent and incorrect, but also indistinguishable with the trace alone. A key reason for these ambiguities is that the execution trace is almost unstructured—there are no pre-defined ways to split the trace to into segments that correspond to loop iterations.

SHEAR infers the unique correct loop structure using *speculative manipulation*. To do this, SHEAR first identifies potential execution points that correspond to queries over which a loop may iterate. In our example, these queries are q_1 and q_3, each of which retrieved two rows. We illustrate these execution points as question marks in Figure 6-4. For each of these execution points, SHEAR performs three *altered executions* of the program to determine whether the potential loop is valid.

Probing Program Behavior via Speculative Manipulation. SHEAR uses a proxy between the program and the database to relay and manipulate the SQL queries

```
q0
for row in q1:
  q2  #(q4)
  q3  #(q5)
q6
q7
```

(a) Correct Plausible Loop L

```
q0
for row in q1:
  q2  #(q4)
  if not q3: #(q5)
    q6
q7
```

(b) Incorrect Plausible Loop W

```
q0
for row in q1:
  q2  #(q4)
  if not q3: #(q5)
    q6
    q7
```

(c) Incorrect Plausible Loop X

```
q0
for row in q1:
  q2  #(q6)
  if q3: #(q7)
    q4
    q5
```

(d) Incorrect Plausible Loop Y

```
q0
q1
q2
for row in q3:
  q4  #(q6)
  q5  #(q7)
```

(e) Incorrect Plausible Loop Z

Figure 6-3: Plausible loop structures that may produce the example trace

in the database traffic. SHEAR first reuses the original inputs and database contents to start executing the program. When the program issues query q_0, SHEAR faithfully relays the database traffic for this query. Next, when the program issues query q_1, SHEAR strategically alters the SQL query into q_1' before forwarding it to the database. The altered query q_1' retrieves only the first row among the rows that would have been retrieved by the original q_1. The database performs q_1' and retrieves the row as requested, which is then relayed through the proxy back to the program. After this manipulation, SHEAR resumes normal program execution until it terminates. The manipulated execution produces the first *altered trace* that consists of queries q_0, q_1', q_2, q_3, q_6, and q_7. Figure 6-5b illustrates this manipulation.

SHEAR next obtains the second altered trace. SHEAR faithfully relays the database traffic for query q_0. When the program issues query q_1, SHEAR alters it into q_1'', which retrieves only the second row among the rows that would have been retrieved by query

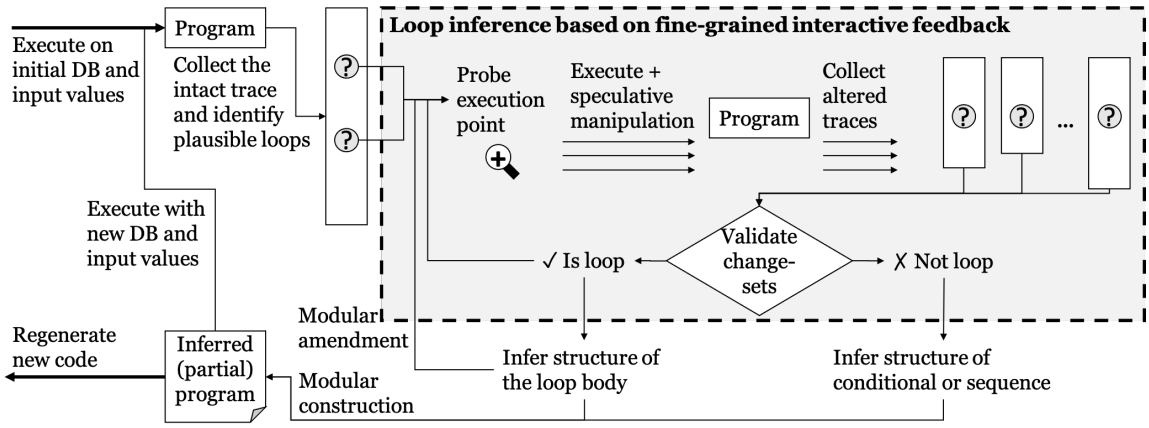


Figure 6-4: SHEAR performs speculative manipulation to infer and regenerate database-backed programs that may contain loop and repetitive structures

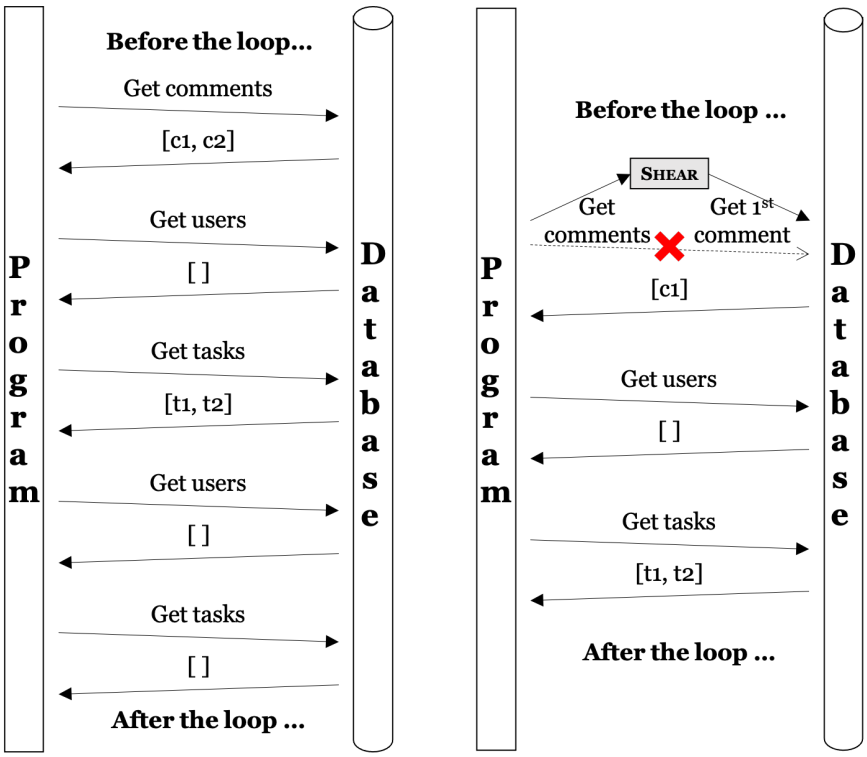
q_1. The database performs q_1'', whose rows are relayed through the proxy back to the program. After this manipulation, SHEAR resumes normal program execution until it terminates. The resulting altered trace consists of queries q_0, q_1'', q_4, q_5, q_6, and q_7. Figure 6-6a illustrates this manipulation.

Finally, SHEAR obtains the third altered trace, where the query q_1 is altered to q_1''' that retrieves both the first and the second rows in q_1. In this example, q_1''' retrieves the same results as q_1. Hence the third altered trace consists of queries q_0, q_1''', q_2, q_3, q_4, q_5, q_6, and q_7. Figure 6-6b illustrates this manipulation.

Change-Set Validation Indicating the Presence of a Loop at q_1. SHEAR compares these three altered traces to determine if a loop iterates over the two rows retrieved by query q_1. Note that queries q_0, q_1', q_1'', and q_1''' are produced before any iterations of the hypothetical loop.

SHEAR first compares the lengths of the three altered traces to calculate the number of queries that would be produced by the subprogram after the hypothetical loop ends. This number is calculated by adding up the lengths of the first two altered traces after the hypothetical loop location (q_1' or q_1''), then subtracting with the length of the third altered trace after the hypothetical loop location (q_1'''). In this example, this number is 2. Figure 6-7 illustrates this comparison.

SHEAR uses this number to infer queries that would be produced by hypothetical

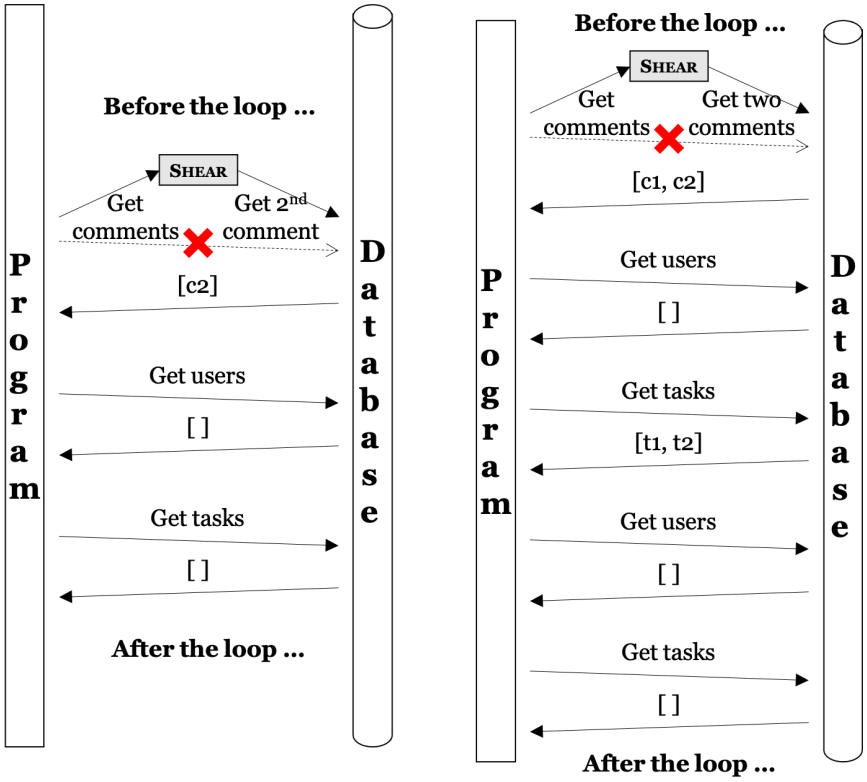


(a) Unaltered trace (b) First altered trace

Figure 6-5: Shear alters the database traffic during program execution to infer loops structures from the example trace

loop iterations. Specifically, Shear removes from each altered trace the last 2 queries and the queries before the hypothetical loop. Remaining queries in the first altered trace are q_2 and q_3, which would be produced by the first hypothetical loop iteration. Remaining queries in the second altered trace are q_4 and q_5, which would be produced by the second hypothetical loop iteration. Remaining queries in the third altered trace are q_2, q_3, q_4, and q_5, which would be produced by both first two hypothetical loop iterations.

Shear then uses these results to check if the hypothetical loop is valid. In this example, the queries produced by the first hypothetical iteration (q_2 and q_3) comprise a strict prefix of the queries produced by both of the first two hypothetical iterations (q_2, q_3, q_4, and q_5). Also, the last 2 queries in all three altered traces are identical (q_6 and q_7). Based on these observations, Shear determines that the program behavior is consistent with the existence of a hypothetical loop. Shear therefore determines



(a) Second altered trace

(b) Third altered trace

Figure 6-6: SHEAR alters the database traffic during program execution to infer loops structures from the example trace (continued)

that a loop indeed iterates over the two rows retrieved by query q_1.

Change-Set Validation Indicating the Absence of a Loop at q_3. Because query q_3 also retrieved two rows during execution, there can potentially be a loop that iterates over the two rows retrieved by query q_3 (Plausible Loop Z). To determine whether this loop exists, SHEAR alters the database traffic for query q_3 to obtain three altered traces. The first altered execution alters query q_3 into query q_3' which retrieves only the first row in query q_3. The resulting altered trace consists of queries q_0, q_1, q_2, q_3', q_4, q_5, q_6, and q_7. The second altered execution alters query q_3 into query q_3'' which retrieves only the second row in query q_3. The resulting altered trace consists of queries q_0, q_1, q_2, q_3'', q_4, q_5, q_6, and q_7. The third altered execution alters query q_3 into query q_3''' which retrieves both the first and the second rows in query q_3. The resulting altered trace consists of queries q_0, q_1, q_2, q_3''', q_4, q_5, q_6, and q_7.

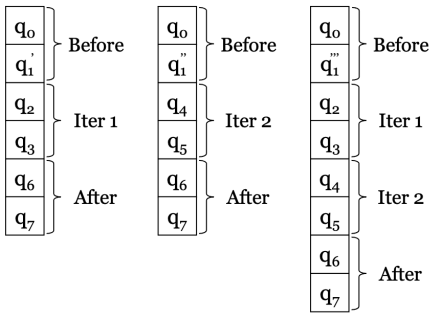
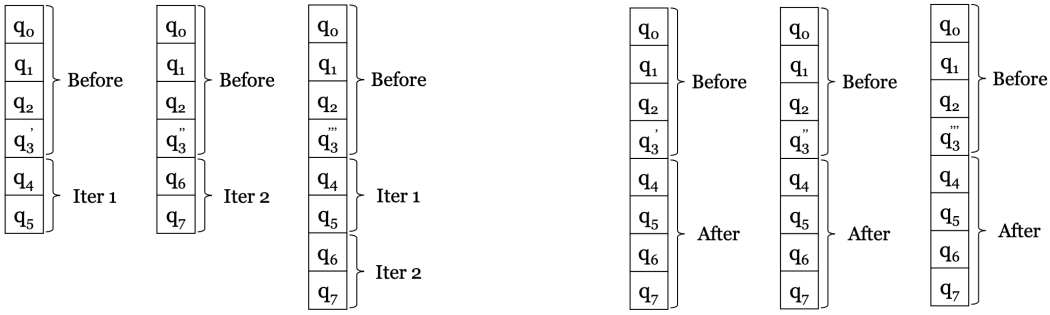


Figure 6-7: SHEAR validates the change-sets for the hypothesis L and concludes that L is correct



(a) Expected if Z were correct (b) Traces that reject Z

Figure 6-8: SHEAR validates the change-sets for the hypothesis Z and concludes that Z is incorrect

SHEAR first compares the lengths of the three altered traces to calculate the number of queries that would be produced by the subprogram after the hypothetical loop ends. This number is calculated by adding up the lengths of the first two altered traces after the hypothetical loop location (q_3' or q_3''), then subtracting with the length of the third altered trace after the hypothetical loop location (q_3'''). In this example, this number is 4. Figure 6-8a illustrates the expected outcome of this comparison.

SHEAR then uses this number to infer the queries that would be produced by the hypothetical loop iterations. Specifically, SHEAR removes the last 4 queries in each altered trace and removes the leading queries up to the hypothetical loop location. For all of the three altered traces, there are no remaining queries. Figure 6-8b illustrates the actual outcome of this comparison.

SHEAR then uses these results to check if the hypothetical loop is valid. In this example, the queries produced by the first hypothetical iteration (no queries) do no comprise a strict prefix of the queries produced by both of the first two hypothetical

iterations (no queries). Based on these observations, SHEAR determines that the program behavior is inconsistent with the existence of a hypothetical loop. SHEAR determines that there are no loops that iterate over the two rows retrieved by query q_3. Hence, SHEAR rules out the incorrect Plausible Loop Z.

Probing and Validating the Loop Body Structure. After SHEAR infers that a loop iterates over query q_1, it manipulates the database traffic again to calculate the loop iteration boundaries. For each row retrieved by q_1, SHEAR obtains an altered trace where q_1 is altered to retrieve only that single row. In this example, because q_1 retrieves only two rows, these altered traces are already obtained earlier when SHEAR detects the existence of the loop.

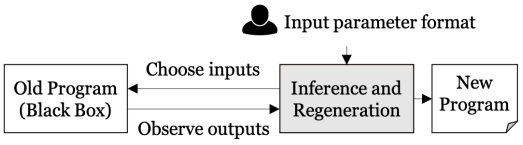
SHEAR compares these traces to first calculate the number of queries in the trace that are generated by the after-loop subprogram. In this case, the after-loop subprogram generates two queries (q_6 and q_7). This result is used to infer the number of queries generated by each loop iteration. In this case, the first iteration generates two queries (q_2 and q_3) and the second iteration generates two queries (q_4 and q_5). Hence, SHEAR rules out the incorrect Plausible Loop W,X,Y and determines that the Plausible Loop L is correct.

Regeneration. After inferring the precise loop and repetition structures, SHEAR proceeds to infer the remaining program structure and regenerate the full program. For the example program, our SHEAR implementation regenerates the code in Figure 6-9. Like KONURE, our current SHEAR implementation regenerates Python code using a standard SQL library to perform database queries (Section 3.1).

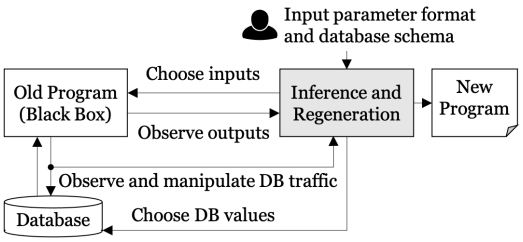
Discussion. SHEAR's program inference algorithm is based on several speculatively manipulated executions of the program per hypothetical loop. The algorithm works precisely with programs that may contain a variety of loop and repetitive structures including nested loops, consecutive loops, loops with conditional statements, and non-loop repetitive queries. These capabilities enable SHEAR to infer and regenerate a wider range of computations than prior work.

```
def example_3 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM tasks WHERE id = :x0", {'
       x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'tasks', 'title'))
    if util.has_rows(s0):
        s1 = util.do_sql(conn, "SELECT * FROM comments WHERE
           task_id = :x0", {'x0': inputs[0]})
        outputs.extend(util.get_data(s1, 'comments', 'content'))
        s1_all = s1
        for s1 in s1_all:
            s2 = util.do_sql(conn, "SELECT * FROM users WHERE id =
               :x0", {'x0': util.get_one_data(s1, 'comments', '
               commenter_id')})
            outputs.extend(util.get_data(s2, 'users', 'name'))
            s3 = util.do_sql(conn, "SELECT * FROM tasks WHERE
               creator_id = :x0", {'x0': util.get_one_data(s1, '
               comments', 'commenter_id')})
            outputs.extend(util.get_data(s3, 'tasks', 'title'))
        s1 = s1_all
        s4 = util.do_sql(conn, "SELECT * FROM users WHERE id = :x0"
           , {'x0': util.get_one_data(s0, 'tasks', 'assignee_id')})
        outputs.extend(util.get_data(s4, 'users', 'name'))
        s5 = util.do_sql(conn, "SELECT * FROM tasks WHERE
           creator_id = :x0", {'x0': util.get_one_data(s0, 'tasks',
            'assignee_id')})
        outputs.extend(util.get_data(s5, 'tasks', 'title'))
    else:
        pass
    return util.add_warnings(outputs)
```

Figure 6-9: SHEAR infers the example program and regenerates code in Python



(a) Generic concept



(b) Specialized for database-backed programs

Figure 6-10: Program inference and regeneration

6.2 Problem Formulation

Program inference and regeneration is the process of observing a program's behavior, inferring aspects of the behavior as a model in a certain domain, and using this inferred model to regenerate a new program (Figure 6-10a). Inferring programs that access an external database involves observing the database interactions [135] (Chapter 3). Shear extends this paradigm with speculative manipulation. That is, Shear not only observes but also manipulates the database interactions (Figure 6-10b). In both Shear and prior work, the technique consists of a DSL and an inference algorithm. We present the DSL below and defer the inference algorithm overview to Section 6.4.1.

6.2.1 DSL for Inferrable Programs

The DSL characterizes the externally observable behavior of database-backed programs that can be inferred. The externally observable behavior consists of the input-output behavior and the database interactions generated during program execution.

We first reiterate key characteristics of the DSL in prior work [135]: (1) each statement performs an SQL query to retrieve rows from the database, (2) the retrieved rows determine the control flow, and (3) the data flow is largely visible in the database traffic.

We next present the Shear DSL. Figure 6-11 presents the DSL for database-backed programs that can be inferred and regenerated by Shear. A program consists of sequences (Seq), conditionals (If), or loops (For). Each Query statement performs

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

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

Figure 6-11: Grammar for the SHEAR DSL

an SQL `select` operation that retrieves data from the database. The query stores the retrieved data in a unique variable (y) for later use. An If statement tests if its Query retrieves empty or nonempty data. A For statement iterates over the rows in its Query. For loops may be nested. The SHEAR DSL is the set of programs $\mathcal{S} \subset$ Prog where the two branches of any If statement start with queries with different skeletons.[1] This DSL expands expressiveness over prior work due to SHEAR's speculative manipulation (Section 7.1.3).

This DSL targets database applications with commands that retrieve data from an external database and, therefore, restricts SQL queries to only `select` statements. One common scenario here is that the commands implement an Internet-accessible, database-backed application. The purpose of the application is to provide read access to the database to remote users who access the database via the Internet.

We highlight the following characteristics of SHEAR's target domain of computations as the key enablers of its loop inference capabilities: (1) There are strong boundaries between components. Specifically, the interface between the program and the external database serves as a strong boundary for SHEAR to observe, probe, and learn information. (2) The component interactions can be altered transparently during program execution. Specifically, the database interactions are altered by SHEAR during program execution without causing runtime errors or crashes. (3) Loops in

[1] As in prior work, this restriction is designed merely to enhance performance distinguishing Seq and If statements and does not affect the correctness of the algorithms.

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

Figure 6-12: Contexts for programs in the DSL

the program iterate over collections that move across component boundaries. Specifically, loops in the DSL iterate over the rows retrieved from the database. After SHEAR alters a database query, the retrieved data changes accordingly, which in turn triggers different behaviors in the remaining program. (4) The execution of each loop iteration generates a deterministic and nonempty trace. In particular, the loop body in the DSL is always nonempty. (5) Each loop iteration operates locally on a single element of the collection. Specifically, each loop iteration in the DSL operates on an individual row among all retrieved rows. This property enables SHEAR to perform change-set validation to infer the loop structures.

6.2.2 Formalizing the Program Behavior

To facilitate discussion, we follow the notation in prior work [135] that characterizes the general domain of database-backed programs and the behavior of such programs. Below, we reiterate the relevant definitions and extend them to work with more sophisticated loop structures, specifically nested loops and loops followed by additional statements.

Definition 6.2.1. A context $\sigma = \langle \sigma_I, \sigma_D, \sigma_R \rangle \in \mathit{Context}$ (Figure 6-12) contains value mappings for the input parameters (σ_I), database contents (σ_D), and results retrieved by database queries (σ_R).

Definition 6.2.2. $\boxed{P}$ denotes the black box executable of a program $P \in \text{Prog}$.

Definition 6.2.3. 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.

Definition 6.2.4. A *loop layout tree* for a program $P \in \text{Prog}$ is a tree that represents information about the execution of loops. Each node in the 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.

Definition 6.2.5. An *annotated trace* is an ordered list of annotated query tuples. Each tuple, denoted as $\langle Q, r, \lambda \rangle$, has three components. The first component is a query $Q \in \text{Query}$. 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. Each path from the root of the loop layout tree to a leaf generates a corresponding annotated trace.

Definition 6.2.6. A *path constraint* $W = (\langle Q_1, r_1, d_1, a_1 \rangle, \ldots, \langle Q_n, r_n, d_n, a_n \rangle)$ consists of a sequence of queries $Q_1, \ldots, Q_n \in \text{Query}$, row count constraints $r_1, \ldots, r_n$, boolean flags $d_1, \ldots, d_n$, and boolean flags $a_1, \ldots, a_n$. Each r_i specifies the range of the number of rows. Each d_i is `true` if a loop iterates over the corresponding retrieved rows and `false` otherwise. Each a_i is `true` if a loop iterates over the corresponding retrieved rows and the path enters the subprogram *after* the loop.

Definition 6.2.7. An annotated trace t 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 a query in W, each row count in t matches a row count constraint in W, and each after-loop status in t matches a flag in W.

Definition 6.2.8. For a program $P \in \text{Prog}$ and a context $\sigma \in \mathit{Context}$, $\sigma \vdash P \Downarrow_{\text{exec}} e$ and $\sigma \vdash P \Downarrow_{\text{loops}} l$ denote evaluating P in σ to obtain a list of query-result pairs e and a loop layout tree l, respectively.

6.3 Probe-and-Validate Cycle for Inferring Loop Structures

Shear takes a database-backed program, infers its functionality, then regenerates a new version of the program with the same inferred functionality (Figure 6-10b). A

Algorithm 11 Infer if a loop iterates over a query by manipulating the database traffic in three program executions

Input: $\boxed{P}$ is the executable of $P \in \mathcal{S}$. σ is a context. k is an integer, denoting a hypothetical loop at k-th query.

Output: Boolean f represents whether a loop is found to iterate over the k-th query in the trace from executing $\boxed{P}$ with σ.

Output: Integer l_α represents the number of queries in the trace produced by the subprogram that follows the inferred loop.

1: **procedure** InferLoopExists($\boxed{P}, \sigma, k$)
2: $s_1 \leftarrow$ ExecAndPick($\boxed{P}, \sigma, k, [1]$) ; $s_2 \leftarrow$ ExecAndPick($\boxed{P}, \sigma, k, [2]$) ; $s_{12} \leftarrow$ ExecAndPick($\boxed{P}, \sigma, k, [1, 2]$)
3: $s'_1 \leftarrow s_1[k+1, \ldots]$; $s'_2 \leftarrow s_2[k+1, \ldots]$; $s'_{12} \leftarrow s_{12}[k+1, \ldots]$
4: $l_\alpha \leftarrow$ Len(s'_1) + Len(s'_2) − Len(s'_{12}) ▷ Len. after hypo. loop
5: $l_1 \leftarrow$ Len(s'_1) $- l_\alpha$ ▷ Len. of the first hypo. iteration
6: $l_2 \leftarrow$ Len(s'_2) $- l_\alpha$ ▷ Len. of the second hypo. iteration
7: $l_{12} \leftarrow$ Len(s'_{12}) $- l_\alpha$ ▷ Len. of both hypo. iterations
8: $\beta_1 \leftarrow s'_1[1, \ldots, l_1]$ ▷ Queries in the first hypo. iteration
9: $\beta_{12} \leftarrow s'_{12}[1, \ldots, l_{12}]$ ▷ Queries in both hypo. iterations
10: $\alpha_1 \leftarrow s'_1[l_1 + 1, \ldots]$ ▷ Queries after the hypo. loop
11: $\alpha_2 \leftarrow s'_2[l_2 + 1, \ldots]$ ▷ Queries after the hypo. loop
12: $\alpha_{12} \leftarrow s'_{12}[l_{12} + 1, \ldots]$ ▷ Queries after the hypo. loop
13: **if** $\alpha_1 = \alpha_2 = \alpha_{12}$ **and** PerfectPrefix(β_1, β_{12}) **then**
14: **return** $\langle$`true`$, l_\alpha\rangle$ ▷ Traces consistent with hypo. loop
15: **end if**
16: **return** $\langle$`false`, Nil$\rangle$ ▷ Traces inconsistent with hypo. loop
17: **end procedure**

key contribution of Shear is manipulating the program's execution on the fly to resolve ambiguities. Shear alters the program's interactions with the environment at precisely chosen execution points. Specifically, Shear infers loops in a database program by speculatively altering the database interactions during program execution. To infer whether there is a loop that iterates over a specific query, Shear removes certain data from the corresponding database interactions and validates how it affects program execution. With a small number of such altered executions, Shear infers whether the execution point contains a loop and, if so, the structure of the loop.

6.3.1 Speculative Manipulation of Database Interactions

Shear manipulates the database interactions through a proxy interposed between the program and the database. In a normal execution of the program, the proxy

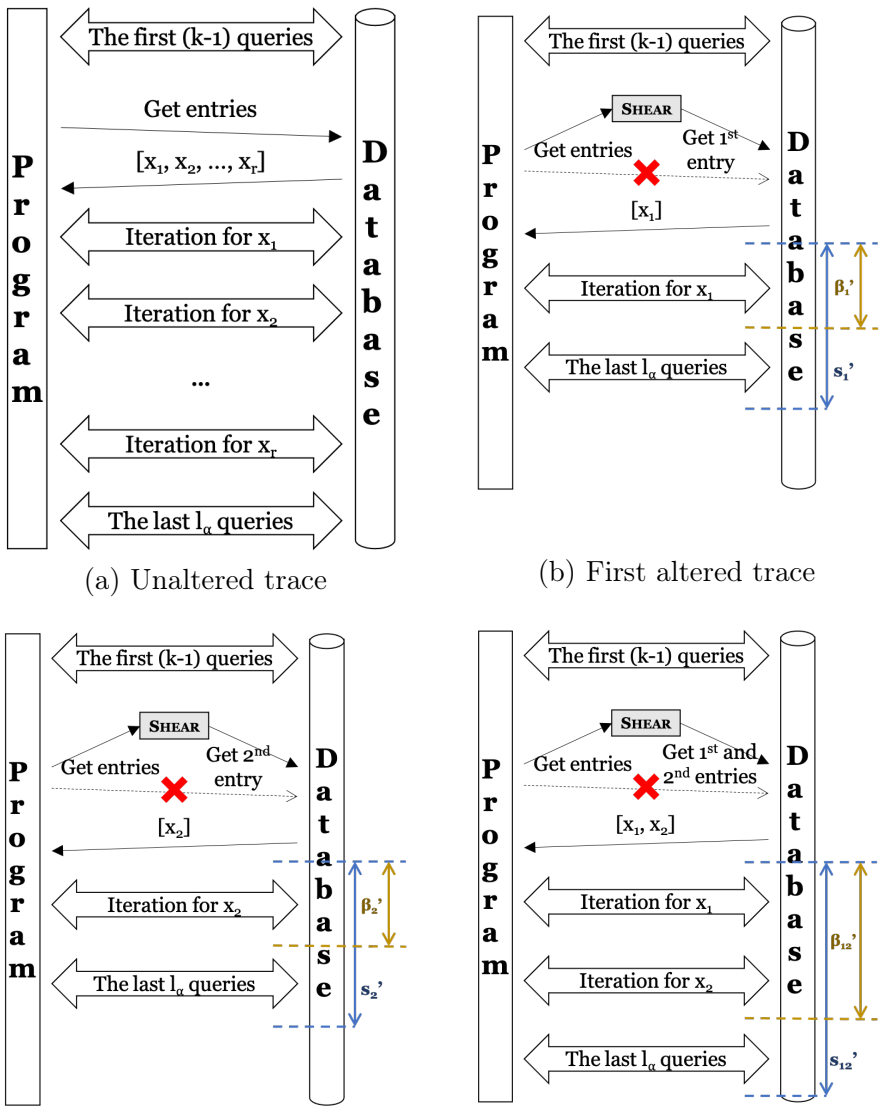


Figure 6-13: Illustration of Algorithm 11 when applied to a true loop that iterates over the k-th query

faithfully relays the interactions between the program and the database. In an execution of the program where SHEAR alters the database interactions, the SHEAR proxy (conceptually) removes certain data from the database interactions while the program runs. Technically, there are two general options to implement this alteration: (1) altering the queries sent from the program to the database or (2) altering the data sent from the database to the program. Our SHEAR implementation uses the first option.

EXECANDPICK. This procedure takes a program, a context, an integer k, and a list of distinct row indices ρ. It executes the program with the context while intercepting the database traffic.

For the first $(k-1)$ queries that the program sends to the database, the proxy relays the traffic faithfully. Up to this point, the collected queries and their retrieved data are identical to what would have been collected from a normal execution of the program.

For the k-th query, the proxy alters the query so that it retrieves only a sub-list of the rows that would have been retrieved if the query were unaltered. The sub-list of rows is specified by the row indices in ρ. Our SHEAR implementation alters the query using standard SQL clauses "`LIMIT`", "`OFFSET`", and "`ORDER BY`". The proxy forwards the altered query to the database, which retrieves (altered) data for the query. The proxy sends the (altered) data back to the program, which processes the data and continues execution accordingly.

After this alteration, the SHEAR proxy continues to relay the rest of the database traffic faithfully until the program terminates. The procedure returns the list of collected SQL queries.

From the program's viewpoint, SHEAR transparently removes certain rows from the data that would have been retrieved. Because SHEAR only removes rows, it only mildly disturbs the execution flow. If the program conforms to the SHEAR DSL, this manipulation does not cause the program to crash. Moreover, in our experiments, this manipulation works reliably with all of our benchmark applications, without

triggering any errors, warnings, or crashes.

6.3.2 Loop Inference Based on Fine-Grained Interactive Feedback

We next present how SHEAR infers loops by performing speculative manipulation and collecting feedback from the altered program executions.

Probing and Validating the Presence of a Loop

For each potential execution point that may contain loops, SHEAR first infers whether the loop exists. If so, SHEAR then infers the structure of each loop iteration. Both steps are achieved through probe-and-validate cycles (Figure 6-4).

INFERLOOPEXISTS. Algorithm 11 infers whether a loop iterates over a specific query during an execution of the program. This procedure takes the program and a context σ. Executing the program with the context σ would produce a trace. The third parameter, k, is an integer query index. The procedure infers if the k-th query in the trace was iterated over by a loop during program execution.

To infer whether a loop iterates over the k-th query, the procedure invokes EXECANDPICK three times. Each invocation executes the program with σ and alters the k-th query. The first execution alters the k-th query to retrieve only the first row among all of the rows that would have been retrieved in an unaltered execution. The second execution alters the query to retrieve only the second row. The third execution alters the query to retrieve only the first two rows. Each execution produces an *altered* list of SQL queries. Because all three executions initially use the same context σ to run the program, the three resulting altered lists are identical up to the $(k-1)$-th query. These three altered lists may differ after the k-th query, depending on the structure of the program.

The INFERLOOPEXISTS procedure then performs location-based change-set validation on these three altered lists, to infer whether the program contains a loop that

Algorithm 12 Infer the loop body structure in terms of the lengths of loop iterations

Input: $\boxed{P}$ is the executable of $P \in \mathcal{S}$. σ is a context. k, r, l_α are integers denoting a loop at k-th query, the number of iterations, and the number of queries after loop, respectively.

Output: List l_β of r integers, where the i-th ($i = 1, \ldots, r$) integer represents the number of queries produced by the i-th loop iteration.

```
1: procedure InferLoopBody(P, σ, k, r, l_α)
2:     l_β ← Empty list
3:     for i ← 1, ..., r do
4:         s_i ← ExecAndPick(P, σ, k, [i])
5:         l_i ← Len(s_i) − k − l_α
6:         Append l_i to l_β
7:     end for
8:     return l_β
9: end procedure
```

iterates over the k-th query. The procedure obtains the three list suffixes starting from the $(k+1)$-th query (line 3). The procedure compares these three list suffixes regarding their lengths and contents. Conceptually, it first assumes there would be a loop that iterates over the specified query. The procedure calculates the number of queries that would have been produced by only the first hypothetical loop iteration (line 5), the number for only the second hypothetical iteration (line 6), and the number for both the first and second iterations (line 7). The procedure then locates the queries that would have been produced by these hypothetical loop iterations (lines 8,9) and by any remaining queries in the program following the hypothetical loop (lines 10,11,12). Figure 6-13 illustrates these calculations. Finally, the InferLoopExists procedure checks if the lengths and contents for these hypothetical iterations are consistent (line 13). The procedure invokes PerfectPrefix with variables β_1 and β_{12}, which represent the queries that would have been produced by the first hypothetical loop iteration and by the first two hypothetical loop iterations, respectively. The PerfectPrefix procedure takes two lists of SQL queries. It returns `true` if and only if (1) the first list is strictly shorter than the second list and (2) the queries in the first list are exactly the same as the corresponding queries at the beginning of the second list.

When a loop is found, the value l_α represents the number of queries in the execution trace that occur after all loop iterations end. Because this value indicates

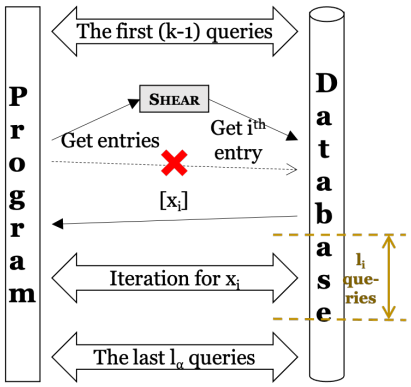


Figure 6-14: Illustration of Alg. 12

exactly where the loop ends, it enables SHEAR to distinguish the loop body and the after-loop subprogram without ambiguity.

This procedure infers that a loop exists if and only if the hypothetical loop is consistent with the three (altered) executions. We show that it detects the potential loop accurately (Section 6.3.3).

Probing and Validating the Loop Body Structure

When a loop has been inferred to be present, SHEAR invokes the INFERLOOPBODY procedure (Algorithm 12) to infer the structure of the loop body. In particular, inferring the loop body structure requires inferring the boundaries for each loop iteration in the execution traces.

INFERLOOPBODY. The INFERLOOPBODY procedure takes a program, a context σ, and three integers k, r, l_α. SHEAR invokes this procedure only when it has inferred that a loop iterates over the r rows retrieved by the k-th query in the trace from executing the program with σ. Because loops in the SHEAR DSL iterate over each row retrieved by the query (Section 6.2.1), the loop has r iterations each accessing one row from the k-th query. The integer l_α is a result from INFERLOOPEXISTS (Algorithm 11) and equals the number of queries in the trace that are produced by the subprogram that follows the detected loop.

The procedure INFERLOOPBODY performs speculative manipulation and location-

based change-set validation to infer the length of each loop iteration in the trace. Specifically, the procedure invokes ExecAndPick for r times. [2] Each invocation executes the program with context σ, but altered so that the k-th query retrieves only one row each time. Each such execution produces a list of (altered) SQL queries. In each list, the first $(k-1)$ queries are unaltered, while the queries after the k-th query are altered. These suffix queries correspond to the queries that would have been produced by one iteration of the loop, followed by l_α queries that are produced by the subprogram in P after the loop. Figure 6-14 illustrates these calculations.

The procedure calculates the lengths of each loop iteration, then returns all of these lengths as a list l_β. Using this list of lengths, it is straightforward to divide the unaltered execution trace into segments that correspond to the individual loop iterations.

Putting it All Together

We present how Shear analyzes any execution trace of the program to infer all of the executed loop structures.

InferLoops. Algorithm 13 starts with an *unaltered* trace from executing the program. For each query in the trace, it invokes InferLoopExists (Algorithm 11) to infer if a loop iterates over the query. If a loop is inferred to exist, InferLoops invokes InferLoopBody (Algorithm 12) to identify loop iteration boundaries in the trace. Both of these procedures leverage Shear's proxy to perform speculative manipulation. The InferLoops procedure finally invokes BuildLLTree to construct a tree that represents the structure of all loops inferred.

BuildLLTree. The BuildLLTree procedure takes a list of query-result pairs e, along with a list L of all loops inferred in e. When there are m inferred loops, the j-th loop iterates over the k_j-th query with iteration lengths $l_{\beta j}$. These lengths indicate the location of the queries in e that are generated by each loop. When there are no

[2] A straightforward optimization is to avoid repeatedly executing the program with the same context and alterations.

Algorithm 13 Infer all loops

Input: $\boxed{P}$ is the executable of $P \in \mathcal{S}$. σ is a context. e is the list of query-result pairs obtained from executing P with σ.
Output: Loop layout tree constructed from e.

```
 1: procedure InferLoops(P, σ, e)
 2:     L ← Empty list
 3:     (Q_1, r_1), ..., (Q_n, r_n) ← e
 4:     for k = 1, ..., n do
 5:         if r_k < 2 then continue end if
 6:         ⟨f, l_α⟩ ← InferLoopExists(P, σ, k)
 7:         if not f then continue end if
 8:         l_β ← InferLoopBody(P, σ, k, r_k, l_α)
 9:         Append ⟨k, l_β⟩ to L
10:     end for
11:     return BuildLLTree(e, L)
12: end procedure
```

nested loops, the queries generated by different loops do not overlap. On the other hand, when there are nested loops, the queries generated by an inner loop are a subset of the queries generated by the outer loop. The procedure constructs a loop layout tree that represents the structure of all loops inferred in e. The procedure builds subtrees bottom-up, first building the subtrees for the last and innermost loops.

Programs in the Shear DSL have the following property: When a loop iterates over the k-th query and when Shear deletes the database traffic from the k-th query, the altered execution traces are the same as the original unaltered trace for all of the queries that do not belong to this loop. Any such altered trace differs from the unaltered trace only by lacking the queries that belong to certain iterations of this loop. As a result, the InferLoops procedure is able to precisely identify the structures of all loops that iterated at least twice (see Section 6.4.1) in any unaltered trace. These algorithms work well for programs that may contain multiple loops, nested or otherwise.

6.3.3 Soundness of the Loop Inference Algorithm

We first outline a soundness proof for the loop inference algorithm and then discuss the intuition.

Definition 6.3.1. For program $P \in \text{Prog}$ and context $\sigma \in \mathit{Context}$, $\sigma[\mapsto_{:k} P]$ denotes the context after evaluating P in σ for k queries. $\textsc{Len}_{\text{exec}}(P, \sigma)$ denotes the length of the trace from evaluating P in σ, that is, $\textsc{Len}_{\text{exec}}(P, \sigma) = \textsc{Len}(e)$ where $\sigma \vdash P \Downarrow_{\text{exec}} e$.

Definition 6.3.2. For program $P \in \text{Prog}$ and list of query-result pairs e, $P - e = P'$ denotes the remaining subprogram after consuming P with e.

Definition 6.3.3. For program $P \in \text{Prog}$ and context $\sigma \in \mathit{Context}$, a loop iterates over the k-th query for r_k times if the following hold for some $P_1, P_2 \in \text{Prog}$: $\sigma \vdash P \Downarrow_{\text{exec}} (Q_1, r_1), \ldots, (Q_n, r_n)$, $1 \leq k \leq n$, and $P - (Q_1, r_1), \ldots, (Q_{k-1}, r_{k-1}) =$ `for` Q_k `do` P_1; P_2. In this case, $\sigma \vdash_{k:} P \Downarrow_{\text{before}} e$ denotes the list of query-result pairs e produced by the subprogram before the loop. $\sigma \vdash_{k:} P \Downarrow^{i}_{\text{iter}} e_i$ denotes the list of query-result pairs e_i obtained from evaluating the i-th iteration of the loop ($i = 1, \ldots, r_k$). $\sigma \vdash_{k:} P \Downarrow_{\text{after}} e$ denotes the list of query-result pairs e produced by the subprogram after the loop. $\Lambda(P, \sigma, k) = [e_1, \ldots, e_{r_k}]$ denotes the list of lists of query-result pairs produced by each of the r_k loop iterations. When no loops iterate over the k-th query, $\Lambda(P, \sigma, k) = \text{NotLoop}$.

Definition 6.3.4. For program $P \in \text{Prog}$, context $\sigma \in \mathit{Context}$, and integer k, $P[k]_\sigma$ denotes the k-th query evaluated when executing P in σ.

Definition 6.3.5. For program $P \in \text{Prog}$, context $\sigma \in \mathit{Context}$, integer k where a loop iterates over the k-th query, and list of integers $\rho_1, \ldots, \rho_m$, $\sigma \vdash_{k:} P \Downarrow^{\rho_1, \ldots, \rho_m}_{\text{alter}} e$ denotes the list of query-result pairs e produced by an altered evaluation of P in σ where the loop that iterates over the k-th query performs only the iterations $\rho_1, \ldots, \rho_m$.

Proposition 6.3.6. For program $P \in \mathcal{S}$, context $\sigma \in \mathit{Context}$, and integer k, if $\Lambda(P, \sigma, k) = [e_1, \ldots, e_r]$, $\sigma \vdash_{k:} P \Downarrow^{\text{Nil}}_{\text{alter}} (Q_1, r_1), \ldots, (Q_n, r_n)$, and $\textsc{ExecAndPick}(\boxed{P}, \sigma, k, [\,]) = [q_1, \ldots, q_{n'}]$, then we have $n = n'$ and q_i corresponds to Q_i for all $i = 1, \ldots, n$.

Proposition 6.3.7. For program $P \in \mathcal{S}$, context $\sigma \in \mathit{Context}$, integer k, and list of distinct integers $\rho_1, \ldots, \rho_m$ ($m > 0$), if $\Lambda(P, \sigma, k) = [e_1, \ldots, e_r]$, $1 \leq \rho_1 < \ldots < \rho_m \leq r$, $\sigma \vdash_{k:} P \Downarrow^{\rho_1, \ldots, \rho_m}_{\text{alter}} (Q_1, r_1), \ldots, (Q_n, r_n)$, and $\textsc{ExecAndPick}(\boxed{P}, \sigma, k, [\rho_1, \ldots, \rho_m]) = [q_1, \ldots, q_{n'}]$, then we have $n = n'$ and q_i corresponds to Q_i for all $i = 1, \ldots, n$.

Theorem 8. For any program $P \in \mathcal{S}$, context $\sigma \in \mathit{Context}$, and integer k, if we have $\sigma \vdash P \Downarrow_{\text{exec}} (Q_1, r_1), \ldots, (Q_n, r_n)$, $1 \leq k \leq n$, $r_k \geq 2$, and $\Lambda(P, \sigma, k) = [e_1, \ldots, e_{r_k}]$, then we have $\textsc{InferLoopExists}(\boxed{P}, \sigma, k) = \langle \texttt{true}, \textsc{Len}(e') \rangle$ where $\sigma \vdash_{k:} P \Downarrow_{\text{after}} e'$.

Proof Sketch. By induction on k and the derivation of loop body. □

Theorem 9. For any program $P \in \mathcal{S}$, context $\sigma \in \mathit{Context}$, and integer k, if $\sigma \vdash P \Downarrow_{\text{exec}} (Q_1, r_1), \ldots, (Q_n, r_n)$, $1 \leq k \leq n$, $r_k \geq 2$, and $\Lambda(P, \sigma, k) = \text{NotLoop}$, then $\textsc{InferLoopExists}(\boxed{P}, \sigma, k) = \langle \texttt{false}, \text{Nil} \rangle$.

Proof Sketch. The proof performs a case analysis of whether a subprogram P' (see below) references Q_k and, if so, where is the first reference. Here P' satisfies $P - (Q_1, r_1), \ldots, (Q_{k-1}, r_{k-1}) = P'$. □

Theorem 10. For any program $P \in \mathcal{S}$, context $\sigma \in \mathit{Context}$, and integer k, if $\Lambda(P, \sigma, k) = [e_1, \ldots, e_r]$ and $\sigma \vdash_{k:} P \Downarrow_{\text{after}} e'$, then $\textsc{InferLoopBody}(\boxed{P}, \sigma, k, r, \textsc{Len}(e')) = [\textsc{Len}(e_1), \ldots, \textsc{Len}(e_r)]$.

Theorem 11. For any program $P \in \mathcal{S}$ and context $\sigma \in \mathit{Context}$, if $\sigma \vdash P \Downarrow_{\text{exec}} e$ and $\sigma \vdash P \Downarrow_{\text{loops}} l$, then $\textsc{InferLoops}(\boxed{P}, \sigma, e) = l$.

These results indicate that Shear is guaranteed to infer loops precisely and return the correct loop layout trees for any program in the DSL. The algorithm works well with any context for (and any initial execution of) the program, regardless of whether the program contains repetitive queries or not. Loop inference is based primarily on how the program behavior changes when the Shear proxy removes certain iterations of a hypothetical loop. The loop body boundaries are determined by how the trace changes when the proxy removes all but one loop iteration.

The most important parts of the proof are Theorem 8 and Theorem 9. Shear needs only several manipulated executions here because Shear works with loops that iterate over collections of rows and whose iterations each operates on one individual row (Section 6.2.1). Intuitively, the loop body can be expressed equivalently as a function that depends only on (1) the one row iterated over by the current loop iteration and (2) the context preceding the loop. When Shear probes an execution point that contains a loop, the resulting altered traces have an unambiguous way to

divide into segments that correspond to the database interactions produced before the loop, produced by each loop iteration, and produced after the loop (Figure 6-13). The segments before and after the loop will be the same across all three altered traces. The segments produced by the loop body in the third altered trace (consisting of two iterations) will be the same as the concatenation of the segment produced by the loop body in the first altered trace (consisting of the first iteration) and the segment produced by the loop body in the second altered trace (consisting of the second iteration). Conversely, when SHEAR probes execution points that are not loops, the resulting altered traces will fail the change-set validations. Such feedback enables SHEAR to infer loops correctly.

6.4 Modular Constructive Inference of Full Program Structure

SHEAR incorporates its loop inference algorithm into a prior technique [135] for the inference and regeneration of database-backed programs (Section 6.2). With speculative manipulation, SHEAR brings in the capability to infer sophisticated loop and repetitive structures.

6.4.1 Modular Inference of Program Constructs

A basic principle in both SHEAR's and prior work's full program inference algorithms is, at each step, to correctly and unambiguously infer a construct in the target program. Technically, this modular construction is performed by expanding a nonterminal symbol in a sentential form in the DSL, that is, a partially expanded abstract syntax tree (AST). This principle enables the algorithms to efficiently infer the program structure and to detect when they have obtained a complete program, even when working with infinite program search spaces.

To preserve this principle in the absence of the ability to alter the program executions, prior work imposes restrictions on the DSL that it can infer. In contrast,

SHEAR preserves this principle while significantly expanding the range of programs that it can infer. The reason for this capability is that, conceptually, speculative manipulation enables SHEAR to break dependencies that otherwise exist across different database queries that retrieve overlapping data. We consider this capability as a fundamental strength of SHEAR.

Preliminaries: Inference Algorithm

We first summarize the algorithm in prior work that infers programs that consist mainly of sequences and conditionals, with limited loops [135]. The inference algorithm uses an SMT solver to generate useful inputs and database values with which to execute the original program. As the algorithm executes the program, it observes the program behavior in terms of the database interactions and the outputs. Based on this observation, the algorithm updates a hypothesis of the inferred program structure as a partially expanded AST. If the updated hypothesis still contains uncertainty, the algorithm resolves this uncertainty by using the SMT solver again to generate new inputs and database values that distinguish different hypotheses. The algorithm recursively expands nonterminal symbols in the inferred program's AST.

To facilitate discussion, we follow the naming convention in prior work and reuse several helper procedures for program execution and solver invocation.

INFER. The entry point to the inference algorithm. It takes an executable $\boxed{P}$ and configures an initial context σ where all database tables are empty and the input parameters are distinct. It invokes GETTRACE on $\boxed{P}$ to obtain an initial annotated trace t. This trace t is used to invoke INFERPROG, which infers the program recursively.

INFERPROG. This procedure infers programs in the SHEAR DSL that may contain sophisticated loop and repetitive structures, including nested loops and after-loop subprograms. This procedure recursively explores all relevant paths through a hypothetical DSL program and resolves Prog nonterminals as they are encountered. The

procedure takes as parameters the executable $\boxed{P}$ and an annotated trace. To resolve a Prog nonterminal, SHEAR examines three annotated traces t_0, t_1, and t_2. In the executions that generated t_0, t_1, and t_2, Q retrieves zero rows, at least one row, and at least two rows, respectively. SHEAR encodes these requirements into path constraints by invoking MAKEPATHCONSTRAINT. SHEAR then obtains the satisfying traces (if they exist) by invoking SOLVEANDGETTRACE. These traces are then used to expand the Prog symbol into one of four statements: ϵ, Seq, If, or For.

GETTRACE. This procedure executes the program, collects a trace, detects loops by invoking INFERLOOPS, and returns an annotated trace that satisfies a path constraint.

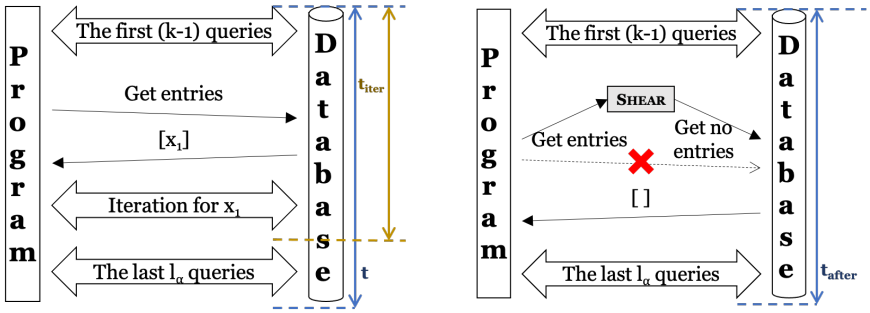
MAKEPATHCONSTRAINT. This procedure takes a trace prefix s_1, a query Q, an integer i, and two lists of boolean flags d and a. The procedure constructs a path constraint, W_i, which specifies that the program should execute down the same path as s_1, then perform Q and retrieve i rows.

SOLVEANDGETTRACE. This procedure takes an executable $\boxed{P}$ and a path constraint W. The procedure solves for a context σ that enables $\boxed{P}$ to produce a trace t that satisfies W. If satisfiable, the procedure invokes GETTRACE to execute the program and obtain a satisfying trace.

Modular Construction of Loops Newly Inferred

The wider range of computations captured by the SHEAR DSL gives rise to challenging loop-related ambiguities that go beyond the scope of prior work (Section 7.1). We briefly describe how SHEAR reasons about potential For statements as a part of the INFERPROG procedure and present more detail in the supplementary material.

SHEAR invokes INFERLOOPS with the unaltered trace t_2 to infer whether the execution point Q contains a loop. (1) If INFERLOOPS infers that the execution point Q does not have loops, SHEAR then proceeds to infer whether Q was generated by an If statement or a Seq statement. This scenario is handled similarly as in prior



(a) Unaltered trace (b) Altered trace

Figure 6-15: Infer the boundaries of the only iteration

work. (2) On the other hand, if InferLoops infers that a loop iterates over the rows retrieved by Q, Shear updates its hypothesis to represent that Q was generated by a For statement. In this case, the InferProg procedure will recursively infer the subprograms inside the For statement. Before entering the recursion, however, it first obtains two additional annotated traces. The trace t_{iter} is an annotated trace whose query Q retrieves at least two rows and whose suffix is generated by the loop body. The trace t_{after} is an annotated trace whose suffix is generated not by the loop body, but by the after-loop subprogram. The InferProg procedure then uses t_{iter} and t_{after} to recursively infer the loop-body subprogram and the after-loop subprogram, respectively. We note that most of the fine-grained information collected during this process is made possible by speculative manipulation.

Modular Inference of Loops that Previously Iterated Only Once

A complication is when a loop iterated only once during an execution of the program. This complication arises from the ability of the Shear DSL to capture programs whose loops are not the last statements. These programs may have statements after loops and even multiple consecutive loops, which are important computations that go beyond the scope of prior work.

The complications are as follows. When a loop iterated only once, the InferLoops procedure does not immediately determine where the (only) loop iteration ends in the provided list of query-result pairs. The resulting loop layout tree there-

fore does not characterize this loop. When SHEAR traverses this tree to generate annotated traces, at least one resulting trace contains both the queries generated by the loop body (which iterated only once) and the queries generated after the loop. These traces, if untreated, would cause the INFERPROG procedure's recursive steps to diverge from the structure of the hypothetical program's AST.

SHEAR solves these complications by constructing two new traces based on the previous trace where the loop iterated only once. SHEAR performs speculative manipulation to identify the boundary of the (only one) loop iteration in the trace. Specifically, SHEAR first matches a trace t against the known Prog nonterminals that have already been inferred. If a query Q is known to be generated by a For statement but retrieved only one row in t, then the corresponding loop iterated only once. In this case, SHEAR invokes EXECANDPICK with an empty list ρ. In the altered execution, the query Q is altered to retrieve zero rows. The corresponding loop iterates for zero times and continues execution after the loop. The resulting trace t_{after} contains only the after-loop queries, without any loop-body queries. Next, SHEAR uses t_{after} to locate the boundary of the loop body in t and discards all of the subsequent queries. The resulting trace t_{iter} contains only the loop-body queries, without any after-loop queries. Figure 6-15 illustrates these calculations.

This way, even though the loop iterated only once in the original trace t, SHEAR is able to update its hypothesis with the correct loop structure. SHEAR does so by discarding t and replacing it with two new traces t_{iter} and t_{after} that correspond correctly to the loop body and the after-loop subprogram, respectively.

6.4.2 Soundness of the Full Program Inference Algorithm

The soundness of this algorithm builds on our key results in Section 6.3.3.

Theorem 12. For any program $P \in \mathcal{S}$, INFER($\boxed{P}$) and P are identical except for the use of different but equivalent origin locations.

Proof Sketch. The proof is by induction on the productions of P. Because the regeneration algorithm extends from prior work (Section 6.4.1), our general proof structure

is similar to that of prior work [138]. Our main modifications are (1) replacing the loop detection theorem in prior work with our Theorem 11 and (2) adding a branch to the structural induction proof of InferProg to account for the second subprogram in For statements in our more expressive DSL. □

Chapter 7

Experimental Evaluation of Shear

We evaluate our loop inference technique with the following research questions:

- **RQ1: Ability to Infer and Regenerate Benchmark Programs.** How does Shear compare with existing techniques in the ability to infer applications that access databases?

  We used Shear to infer the commands in several open-source database-backed applications. Compared to existing techniques, Shear infers a wider range of loop and repetitive structures (Section 7.1).

- **RQ2: Scalability.** How does our technique scale with more complex loop structures?

  Our results indicate that Shear scales well for most dimensions of program complexity. The only dimension for which Shear does not scale well is the number of layers of nested loops (Section 7.2).

- **RQ3: Speculative Manipulation Versus Enumerative Search.** How does our technique compare with enumerative search?

  In our experiments, the Shear inference times grow asymptotically slower than the search space sizes do with most dimensions of program complexity. The only dimension along which Shear is asymptotically slower is the number of loop iterations (Section 7.3).

We performed experiments on Ubuntu 16.04 virtual machines with 6 cores and 4 GB memory. The host machine uses a processor with 6 cores (2.9 GHz Intel Core i9) and has 32 GB 2400 MHz DDR4 memory. Our SHEAR implementation uses Python 3.7.9 (PyPy 7.3.3) and Z3 4.6.0.

7.1 RQ1: Ability to Infer and Regenerate Benchmark Programs

We evaluate SHEAR on a set of benchmark applications and a synthetic test suite. Each application has several commands; SHEAR infers one command at a time. Each command takes input parameters, performs SQL queries accordingly, and outputs some of the retrieved data.

7.1.1 Benchmark Applications

Our benchmark applications include:

- **RailsCollab Project Manager:** RailsCollab [8] is an open-source project management and collaboration tool, built with Ruby on Rails, with over 250 stars on GitHub. The source code contains 11944 lines of Ruby, HTML, CSS, and JavaScript. RailsCollab maintains multiple task lists, tasks, milestones, time records, and messages. RailsCollab retrieves data from 24 relevant tables with 270 columns. Its commands enable users to navigate these contents.

- **Kanban Task Manager:** Kanban [11] is an open-source task management system, built with Ruby on Rails, with over 600 stars and 200 forks on GitHub. The source code contains 1653 lines of JavaScript, SASS, Ruby, and HTML. Kanban maintains boards. Each board may contain multiple lists. Each list may contain multiple cards, each of which may have comments. Kanban retrieves data from 4 relevant tables with 42 columns. Its commands enable users to navigate boards, lists, cards, and comments.

- **Todo Task Manager:** Todo [9] is an open-source task-tracking tool, built with Ruby on Rails, with over 100 stars and 180 forks on GitHub. The source code contains 1340 lines of HTML, JavaScript, Ruby, CSS, and SASS. Todo maintains multiple lists. Each list may contain multiple tasks. Todo retrieves data from 2 relevant tables with 10 columns. Its commands enable users to navigate lists and tasks.
- **Fulcrum Task Manager:** The same Fulcrum [3] application as in Section 5.1.1.
- **Kandan Chat Room:** The same Kandan [5] application as in Section 5.1.1.
- **Enki Blogging Application:** The same Enki [2] application as in Section 5.1.1.
- **Blog:** The same Blog [4] application as in Section 5.1.1.
- **Student Registration:** The same Student [7] application as in Section 5.1.1.
- **Synthetic:** A set of synthetic Python programs with repetitions, nested loops, and consecutive loops designed to challenge other loop inference techniques.

Five of these applications – RailsCollab, Kanban, Fulcrum, Kandan, and Enki – are studied in a recent survey [173]. We identified the Todo Task Manager from popular Ruby on Rails projects on GitHub. Five of the applications – Fulcrum, Kandan, Enki, Blog, and Student – were used in the evaluation of Konure. A copy of the source code for the benchmark applications is available at [10]. The synthetic test suite highlights the capability of Shear. The source code for the synthetic test suite is available in Appendix C.

Characteristics of Loop and Repetition Structures. Our benchmarks have structures as follows.

- **Non-Loop Repetitive Queries:** Codes "R" and "r" denote that a command contains non-loop repetitive queries that violate the heuristic assumptions in other loop inference systems. Specifically, the code "R" denotes that the command may generate an execution trace that contains non-adjacent repetitions. The code "r" denotes generating adjacent repetitions.

- **Control Structures in the Loop Body:** The code "C" denotes that a command has a loop whose loop body contains control structures such as conditional statements or loops.

- **Nested Loops:** The code "N_i" denotes that a command contains i layers of nested loops when the command's externally observable behavior is expressed in the SHEAR DSL.

- **After-Loop Subprograms:** The code "A" denotes that a command contains after-loop subprograms. The code "A_i" denotes that the command contains i loops, one after the other.

Commands with code "R" are out of the scope of KONURE, Kobayashi's algorithm [99], and WebRobot [64]. Commands with code "r" are out of the scope of KONURE, DaViS [110], Kobayashi's algorithm, and WebRobot. Commands with code "C" are out of the scope of DaViS, Kobayashi's algorithm, and WebRobot. Commands with code "N_i" are out of the scope of KONURE, DaViS, and Kobayashi's algorithm. Commands with code "A" or "A_i" are out of the scope of KONURE.

7.1.2 Results

Experimental Results for Loop Inference. Table 7.1 and Table 7.2 present the results of applying SHEAR to the benchmark commands, along with a comparison with other loop inference techniques. Each row presents a command. A row is shaded if the corresponding command does not have any loop or repetitive structures as expressed in the SHEAR DSL. The first column (**Command**) presents the command name. The next (**Time**) presents the wall-clock inference time. The next (**LoC**) presents the number of lines in the regenerated Python code. The next three (**Q**, **If**, and **For**) present the numbers of SQL, If, and For statements as expressed in the DSL. The next (**Struct**) presents the characteristics of loop and repetitive structures outlined above. The next two (**S** and **Kn**) represent whether the command is supported by SHEAR and KONURE, respectively. The next two (**D** and **Kb**) represent whether the execution

Table 7.1: SHEAR's performance on inferring and regenerating benchmark commands

Command	Time	LoC	Q	If	For	Struct	S	Kn	D	Kb	WR
`get_projects_id_messages`	7203.3s	114	34	9	3	R, C, A_2	✓	✗	✗	✗	✗
`get_projects_id_messages_id`	4326.4s	69	21	5	0	r	✓	✗	✗	✗	✗
`get_projects_id_messages_display_list`	7461.4s	115	35	9	3	R, C, A_2	✓	✗	✗	✗	✗
`get_projects_id_times` (fixed 500 error)	8739.7s	114	38	11	1	R, C, A	✓	✗	✗	✗	✗
`get_projects_id_times_id`	3846.7s	147	55	15	0	R	✓	✗	✓	✗	✗
`get_projects_id_milestones_id`	5544.6s	93	25	6	2	R, r, A	✓	✗	✗	✗	✗
`get_users_id` (RailsCollab) (fixed 500 error)	960.3s	53	12	5	1		✓	✓	✓	✓	✓
`get_api_lists`	1557.2s	55	9	1	3	C, N_3	✓	✗	✗	✗	✗
`get_api_lists_id`	765.5s	54	9	2	2	C, N_2	✓	✗	✗	✗	✗
`get_api_cards`	780.3s	46	8	1	2	C, N_2	✓	✗	✗	✗	✗
`get_api_cards_id`	398.4s	45	8	2	1		✓	✓	✓	✓	✓
`get_api_boards_id`	1329.2s	64	10	2	3	C, N_3, A	✓	✗	✗	✗	✗
`get_home` (Todo)	529.3s	20	5	1	1	C	✓	✓	✗	✗	✗
`get_lists_id_tasks`	565.9s	17	6	1	1	C, A	✓	✗	✗	✗	✗
`get_lists_id_tasks` (fixed 404 error)	742.2s	20	6	1	1	C, A	✓	✗	✗	✗	✗
`get_channels`	3762.2s	53	16	4	2	C	✓	✓	✗	✗	✗
`get_home` (Enki)	757.2s	26	9	1	1	A	✓	✗	✓	✓	✓
`get_archives`	659.4s	21	6	1	1	C, A	✓	✗	✗	✗	✗
`get_admin_posts`	397.5s	17	3	1	1		✓	✓	✓	✓	✓
`get_admin` (trimmed)	773.9s	21	7	0	2	A_2	✓	✗	✓	✓	✓
`liststudentcourses`	171.1s	22	5	2	1		✓	✓	✓	✓	✓
`repeat_2`	82.0s	7	3	0	0	r	✓	✗	✗	✗	✗
`repeat_3`	102.7s	8	4	0	0	R, r	✓	✗	✗	✗	✗
`repeat_4`	122.2s	9	5	0	0	R, r	✓	✗	✗	✗	✗
`repeat_5`	143.0s	10	6	0	0	R, r	✓	✗	✗	✗	✗

Table 7.2: SHEAR's performance on inferring and regenerating benchmark commands (continued)

Command	Time	LoC	Q	If	For	Struct	S	Kn	D	Kb	WR
`nest`	393.8s	15	3	0	2	C, N_2	✓	✗	✗	✗	✗
`after_2`	220.8s	15	4	0	2	R, A_2	✓	✗	✓	✗	✗
`after_3`	328.7s	20	6	0	3	R, A_3	✓	✗	✓	✗	✗
`after_4`	441.1s	25	8	0	4	R, A_4	✓	✗	✓	✗	✗
`after_5`	549.4s	30	10	0	5	R, A_5	✓	✗	✓	✗	✗
`example` (Section 6.1)	293.3s	22	6	1	1	R, A	✓	✗	✓	✗	✗
`get_projects`	295.3s	26	7	2	0		✓	✓	–	–	–
`get_companies_id`	257.3s	28	6	2	0		✓	✓	–	–	–
`get_api_users_current`	41.3s	9	1	0	0		✓	✓	–	–	–
`get_api_users_id`	39.0s	9	1	0	0		✓	✓	–	–	–
`get_home` (Fulcrum)	165.2s	21	5	1	0		✓	✓	–	–	–
`get_projects`	165.4s	21	5	1	0		✓	✓	–	–	–
`get_projects_id`	584.1s	25	8	2	0		✓	✓	–	–	–
`get_projects_id_stories`	366.3s	31	8	3	0		✓	✓	–	–	–
`get_projects_id_stories_id`	363.7s	31	9	3	0		✓	✓	–	–	–
`get_projects_id_stories_id_notes`	364.7s	24	9	3	0		✓	✓	–	–	–
`get_projects_id_stories_id_notes_id`	391.6s	28	10	4	0		✓	✓	–	–	–
`get_projects_id_users`	525.0s	25	8	2	0		✓	✓	–	–	–
`get_channels_id_activities`	1388.2s	49	16	6	0		✓	✓	–	–	–
`get_channels_id_activities_id`	844.3s	25	11	3	0		✓	✓	–	–	–
`get_me`	637.7s	44	8	3	0		✓	✓	–	–	–
`get_users`	1872.4s	67	11	3	0		✓	✓	–	–	–
`get_users_id` (Kandan)	678.7s	44	8	3	0		✓	✓	–	–	–
`get_admin_comments_id`	62.4s	10	1	0	0		✓	✓	–	–	–
`get_admin_pages`	220.7s	13	2	1	0		✓	✓	–	–	–
`get_admin_pages_id`	51.8s	9	1	0	0		✓	✓	–	–	–
`get_article_id`	92.1s	15	2	1	0		✓	✓	–	–	–
`get_articles`	60.8s	8	1	0	0		✓	✓	–	–	–

traces produced by the command are compatible with DaViS and Kobayashi's loop inference algorithms [99], respectively. The next (**WR**) represents whether the loop structures can be supported by an adaptation of the rewrite-based loop synthesis algorithm in WebRobot. Symbols "✓" and "✗" indicate that a command is in the scope and out of the scope of a technique, respectively. Symbol "–" indicates that a command is not applicable for the comparison of loop inference techniques.

Compared to these four other techniques, SHEAR supports a substantially more complex set of loop and repetitive structures. All 25 benchmark commands with loops (whose "For" column is nonzero) are supported by SHEAR. In contrast, only 13 of these commands are supported by at least one of the four other techniques. All 6 benchmark commands without loops but contain repetitive queries (whose "For" column is zero) are supported by SHEAR. In contrast, only one of these commands is supported by at least one of the four other techniques. Overall, 25 benchmark commands are out of the scope of KONURE, 18 commands are out of the scope of DaViS, and 25 commands are out of the scope of both Kobayashi's [99] and WebRobot's [64] algorithms. In contrast, all of these commands are supported by SHEAR. We attribute these differences to SHEAR's capability to precisely disambiguate loops with speculative manipulation.

Experimental Results for Program Inference. We applied SHEAR to infer and regenerate 53 commands in our benchmarks. These commands include 9 from RailsCollab, 7 from Kanban, 3 from Todo, 8 from Fulcrum, 6 from Kandan, 7 from Enki, 2 from Blog, 1 from Student, and 10 from Synthetic. Among these supported commands, 25 (47%) are out of the scope of KONURE (Table 7.1 and Table 7.2).

The wall-clock inference times range between 39–8740 seconds (average 1197 seconds). Among the 25 benchmark commands that are out of the scope of KONURE, SHEAR's inference times range between 82–8740 seconds (average 1909 seconds). Among the 28 benchmark commands that are supported by both SHEAR and KONURE, SHEAR's inference times range between 39–3762 seconds (average 560 seconds). The former commands are often larger and more complex than the latter.

We next compare SHEAR and KONURE on the 21 shared benchmark commands that are used in the evaluation of both systems.[1] The unaltered execution numbers in SHEAR are slightly higher than the corresponding numbers reported in KONURE. We attribute this difference to the different uses of the solver when implementing these two systems. The number of altered executions in SHEAR range between 0–6.3 times (average 1.6 times) that of unaltered executions. We attribute this overhead to SHEAR's algorithm that infers a wider range of loop and repetitive structures.

The number of solver invocations in both systems are roughly in the same range.

SHEAR's inference times are generally shorter than KONURE for commands that required many solver invocations. We attribute this difference to the different uses of the solver when implementing these two systems. SHEAR's inference times are generally longer than KONURE for commands that required many executions. We attribute this difference to the increased number of command executions by SHEAR for inferring more expressive loop and repetitive structures.

Experimental Results for Program Regeneration. For each benchmark command, SHEAR regenerates a Python program. We present all of the regenerated programs in Appendix D. The regenerated programs have between 1–55 (average 9.8) SQL queries, between 0–15 (average 2.3) If statements, between 0–5 (average 0.9) For statements, between 0–50 (average 12.7) lines that generate output, and between 7–147 (average 36.0) total lines of code.

We compared the regenerated programs for the shared commands. For each shared command in Enki, Fulcrum, Kandan, and Student, SHEAR and KONURE infer and regenerate equivalent programs. For each command in Blog, SHEAR and KONURE infer and regenerate equivalent programs except for a minor difference in the scripts for restarting and executing the application.

[1] Our benchmarks contain 7 commands that can be supported by KONURE but were not used in KONURE's evaluation.

Scope. Some of these benchmark applications implement read-only data-retrieval commands[2] that are out of the scope of SHEAR. Six such commands in RailsCollab and two in Kandan retrieve files or folders. One in Kanban and one in Fulcrum retrieves metadata such as session keys and history updates. Four in RailsCollab and one in Kanban iterate over the rows retrieved by an earlier query that does not immediately precede the first iteration of the loop body. One in RailsCollab contains conditional statements that do not depend on whether the preceding query retrieves empty or nonempty. The majority of the remaining data retrieval commands in Enki and RailsCollab involve application-specific calculations such as concatenating multiple input strings, checking whether a datetime is smaller than another, and enumerating a set of activity type strings. SHEAR infers all of the data retrieval commands in Todo, Blog, Student, and Synthetic.

7.1.3 Discussion

Compared to four other techniques, SHEAR supports a wider range of loop and repetitive structures that occur in real-world database-backed applications. When applied on identical benchmarks that are supported by both SHEAR and KONURE, the two infer and regenerate equivalent results, with the main difference being that SHEAR may execute the benchmarks more times. This overhead often becomes less noticeable for larger programs, as the inference algorithm spends a larger portion of the time interacting with the solver. Almost half of our benchmark commands contain loop and repetitive structures that are supported by only SHEAR but not KONURE.

We attribute these improvements to the capability of SHEAR to perform speculative manipulation. Intuitively, speculative manipulation enables SHEAR to eliminate dependencies between constructs that access overlapping sets of database items, which in turn enables SHEAR to more precisely control program behavior and obtain fined-grained interactive feedback. SHEAR exploits this control to infer a larger and

[2]For a Ruby on Rails application, these commands correspond to the routes that handle HTTP GET requests with an `index` action, a `show` action, or an action that displays the current user. We count such routes only if their corresponding actions are implemented and access the database.

more expressive class of programs than KONURE while preserving the advantages of the KONURE inference algorithm, which proceeds by correctly and unambiguously expanding one nonterminal at each step of the algorithm. This capability enables SHEAR to eliminate many of the heuristic restrictions in prior work. It enables the unambiguous inference of a wider range of computations that are out of the scope of prior techniques. The four prior techniques all use repetition-based pattern matching to infer loops from the execution traces of programs. Because there are often multiple program structures that can generate the observed repetitions, these prior techniques all resort to heuristics to deal with the ambiguity. They either limit the loop and repetitive structures they can work with or simply pick one of the multiple nonequivalent candidate program structures.

In particular, the SHEAR DSL is more expressive than KONURE as follows. The KONURE DSL uses heuristics to impose a range of restrictions on the structure of the program—for example, it requires the (unchecked) property that any query that follows a query in the DSL program that may retrieve multiple rows must not have the same query skeleton as any following query. It also requires any loop to be the last statement of the program, that is, KONURE does not allow any other statements to follow after a loop ends. And it does not support nested loops. Because the SHEAR DSL does not have these restrictions, it is much more expressive than the KONURE DSL: (1) The SHEAR DSL supports more general looping constructs, specifically loop nests and intermixed sequences of sequential code, loops, and conditionals. In contrast, the KONURE DSL supports only a single loop at the end of the program with no code after the loop. (2) The SHEAR DSL supports repeated queries. In contrast, the KONURE DSL imposes restrictions on repeated queries. These differences are a result of SHEAR's speculative manipulation, which enables SHEAR to infer more benchmarks: 10 of our open-source commands have code after loops, 4 have nested loops, and 6 have non-loop repetitions—all these commands are supported by SHEAR but not KONURE. SHEAR infers all 31 commands (21 open-source and 10 synthetic) that contain loops or repetitions, while KONURE infers only 6 of them.

```
for n_loops in range(l):
  s = do_sql("SELECT * FROM t1")
  for row in s:
    for n_body in range(b):
      do_sql("SELECT * FROM t0")
```

(a) with l consecutive loops, where each loop body has b queries

```
do_sql("SELECT * FROM t1")
for n_query in range(q):
  do_sql("SELECT * FROM t2")
```

(b) with q non-loop repetitive queries

```
def rec(n_layer):
  if n_layer <= 0:
    do_sql("SELECT * FROM t0")
    return
  s = do_sql("SELECT * FROM t1")
  for row in s:
    rec(n_layer - 1)

rec(n)
```

(c) with n layers of nested loops

Figure 7-1: Programs for testing scalability

7.2 RQ2: Scalability

We empirically evaluate the scalability of Shear's loop inference algorithm along seven dimensions of program complexity. These dimensions are: (a) the number of consecutive loops in the program, (b) the number of iterations executed for a loop, (c) the number of non-loop repetitive queries in the program, (d) the number of queries in the body of a loop, (e) the number of non-loop queries that retrieve multiple rows during execution, (f) the number of both consecutive loops and database rows, and (g) the number of layers of nested loops. The dimension f is designed to simulate the effects of applying Shear to infer full programs, because our database solver often inserts more rows for longer program paths. By definition, the dimension f reflects the combined effects of both dimensions a and b.

Independent Variables. For each dimension of program complexity, we developed a set of synthetic Python programs (along with the associated input and database val-

ues) with various levels of complexity. Each program's externally observable behavior is expressible in the SHEAR DSL. Specifically, we generated the synthetic programs for dimension (a) from Figure 7-1a with $r = 2$, $b = 1$, and varying values for l, (b) from Figure 7-1a with $l = 1$, $b = 1$, and varying values for r, (c) from Figure 7-1b with $r = 8$ and varying values for q, (d) from Figure 7-1a with $r = 2$, $l = 1$, and varying values for b, (e) from Figure 7-1a with $r = 2$, $b = 0$, and varying values for l, (f) from Figure 7-1a with $b = 1$ and varying values for both r and l ($r = l$), and (g) from Figure 7-1c with varying values for n. Here r denotes the number of rows we insert into the table `t1` in the database.

Note that for the complexity dimensions b and f, setting $r = 1$ means inserting only one row into the table `t1`, which causes the loop to iterate only once. Loops that iterated only once are not identified by loop inference alone (Section 6.4.1).

Dependent Variables. For each dimension of complexity and value of the independent variable, we executed the synthetic program with the associated input and database values. We measured the *wall-clock time for executing the program.* This execution produces an unaltered trace. We then used this unaltered trace (along with the executable program) to invoke SHEAR's loop inference algorithm. We measured the *wall-clock time for inferring loops* in this trace.

7.2.1 Results

Figure 7-2 presents the scalability results for SHEAR's loop inference. Each subfigure corresponds to a dimension of complexity. The horizontal axes represent the independent variables. The vertical axes represent the dependent variables in numbers of seconds.

The green circles represent program execution times. The program execution time is positively correlated with the length of the unaltered execution trace. The traces in our experiments contain between 0–48 queries (Figure 7-2a), 2–17 queries (Figure 7-2b), 1–17 queries (Figure 7-2c), 3–33 queries (Figure 7-2d), 0–16 queries (Figure 7-2e), 2–272 queries (Figure 7-2f), and 1–127 queries for 0–6 layers of nested

loops and 255–1023 queries for 7–9 layers (Figure 7-2g).

The blue triangles represent loop inference times. SHEAR correctly infers the unique correct loop structures for all of these traces.

7.2.2 Discussion

In these experiments, the growths of the loop inference times are correlated mainly with the number of altered program executions during loop inference. For the plotted values, the loop inference times scale linearly with the number of loops (Figure 7-2a), the number of loop iterations (Figure 7-2b), and the number of queries retrieving multiple rows (Figure 7-2e). For the plotted values, the loop inference times remain stable with various numbers of non-loop repetitions (Figure 7-2c) and numbers of queries in the loop body (Figure 7-2d) and scale quadratically with the number of both loops and database rows (Figure 7-2f). In theory, the asymptotic growths of the loop inference times are also correlated with the times for executing the program and the times for parsing the execution traces. These two time components both grow linearly along dimensions a–e and quadratically along the dimension f. The loop inference times grow exponentially with increasing layers of nested loops (Figure 7-2g). We attribute this phenomenon to the exponential growth in the lengths of execution traces for deeper nested loops. Overall, SHEAR's loop inference algorithm scales polynomially along the first six dimensions of program complexity.

7.3 RQ3: Speculative Manipulation Versus Enumerative Search

We evaluate the asymptotic efficiency of SHEAR's loop inference algorithm by contrasting the loop inference time against the size of the search space of candidate program structures. We note that many program synthesis techniques are based on enumerative search with pruning [14, 18, 32, 50, 69, 79, 93, 102, 103, 111, 112, 143, 144, 158, 162, 163, 175, 116].

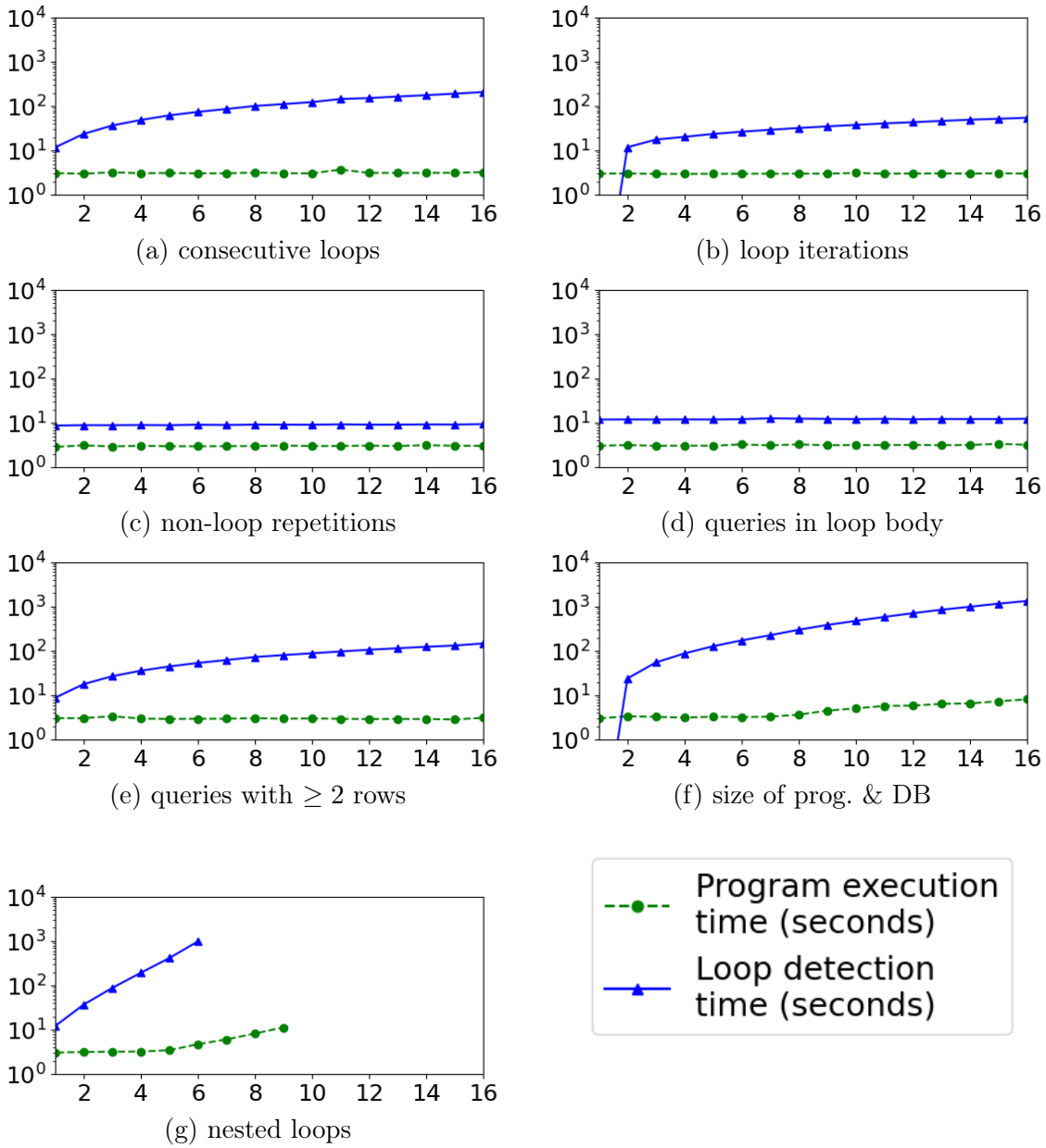


Figure 7-2: Time to infer loops in programs with various measures of complexity

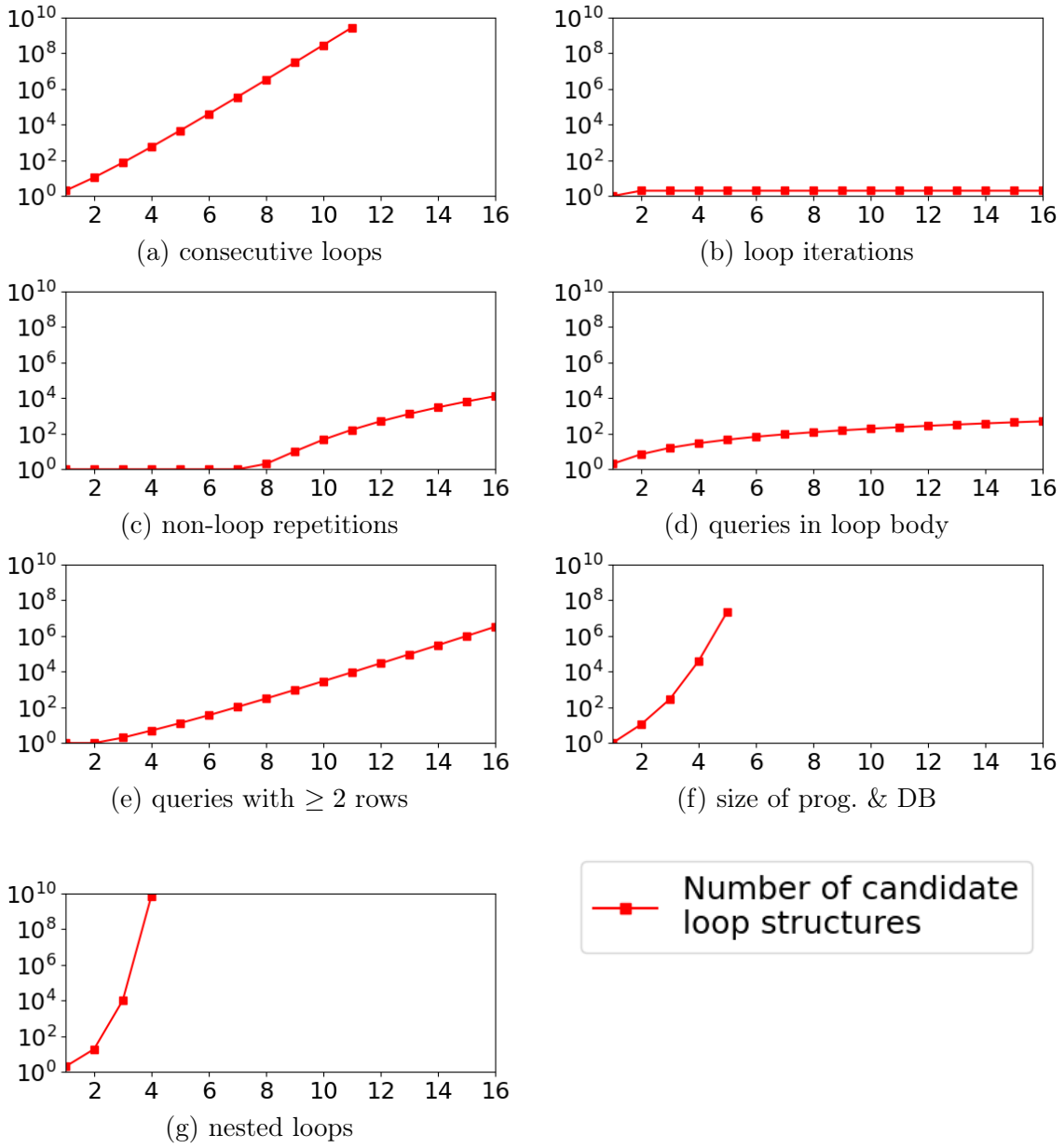


Figure 7-3: Number of candidate loop structures in programs with various measures of complexity

Independent Variables. We conducted two experiments that calculate the search space sizes. The first experiment uses the same set of independent variables as in Section 7.2. The second experiment uses the maximum depth of programs as the independent variable, where depth is defined as the maximum number of SQL queries along any path in the program's AST.

Dependent Variables. The first experiment uses the unaltered traces collected from executing the synthetic programs for the various dimensions of complexity and values of the independent variables. For each trace, we calculated the *number of candidate loop structures* that may generate the trace. The second experiment is based on theoretical calculation. For each maximum program depth, we calculated the *number of candidate AST program structures* under this depth.

7.3.1 Results

Figure 7-3 presents the results for the first experiment. Each subfigure corresponds to one dimension of program complexity. The horizontal axes represent the independent variables. The vertical axes represent the sizes of the search space of candidate loop structures given an unaltered trace. To evaluate the efficiency of SHEAR's loop inference algorithm, we compare the asymptotic growths of the search space sizes against the loop inference times. For the plotted values, the search space sizes grow at least exponentially with increasing numbers of consecutive loops (Figure 7-3a), numbers of queries retrieving multiple rows (Figure 7-3e), and numbers of both loops and database rows (Figure 7-3f). In contrast, the SHEAR loop inference times grow polynomially along these dimensions (Figure 7-2a, Figure 7-2e, and Figure 7-2f). Compared to the loop inference times, the search space sizes also grow asymptotically faster with increasing numbers of non-loop repetitions (Figure 7-3c), numbers of queries in the loop body (Figure 7-3d), and numbers of layers of nested loops (Figure 7-3g). We attribute these differences to SHEAR's ability to infer loops efficiently. The only dimension along which the search space sizes grow slower than loop inference times is the number of loop iterations (Figure 7-3b). We attribute this

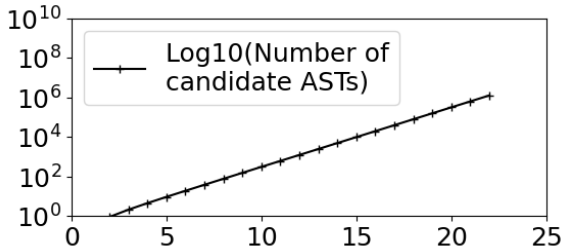


Figure 7-4: Log base 10 of the number of candidate AST structures in programs with various maximum depths

difference to the increased number of program executions required for SHEAR to infer the boundaries for an increased number of loop iterations.

Figure 7-4 presents the results for the second experiment. The horizontal axis represents the maximum depth of a program in the SHEAR DSL. The vertical axis represents the size of the search space of candidate AST structures. Because these numbers are too large for our plotting software, we plot them after taking logarithm with base 10. This search space grows double-exponentially with the maximum program depth.

7.3.2 Discussion

Overall, the asymptotic performance of SHEAR's loop inference algorithm (Figure 7-2) is very efficient compared to the search space of candidate loop structures (Figure 7-3). We attribute this efficiency to the fact that SHEAR's algorithm does not require enumerative search—instead, it is based on manipulating the program executions and comparing the trace lengths. A naïve search space of candidate AST programs may contain programs whose depths are less than or equal to the trace length, up to 272 in our experiments. This search space is way beyond the plotting capability of Figure 7-4 and infeasible to enumerate. In contrast, the SHEAR loop inference algorithm terminates within half an hour on a laptop computer for all of these experiments (Figure 7-2).

7.4 Conclusion

The need to infer loop constructs from observations of program executions has repeatedly arisen in a range of fields. We present a novel dynamic analysis technique that automatically infers loop and repetitive structures from execution traces. The algorithm strategically alters the program's database traffic at precisely chosen execution points to elicit different behaviors, which differ depending on the structure of the underlying program. Results from our implementation highlight the effectiveness of our technique at eliminating many of the limitations present in other systems.

Chapter 8

Conclusion

Human society will continue to develop and rely on software. The goal of my research is to improve software development by automating tasks that currently require substantial manual engineering effort, more specifically by providing a powerful way for developers to express program functionality that adapts flexibly to a variety of contexts. This thesis demonstrates that the program inference and regeneration framework satisfies this goal.

We have demonstrated the feasibility of this framework by implementing it in Konure and Shear. These systems take a gray-box active learning approach. They treat an existing program as a gray box that consists of black-box components that interact with each other, observing the behaviors of the black boxes, controlling the environments in which they operate, inferring models of their core functionality, and finally using the models to regenerate new implementations. The new implementations deliver the same core functionality but are potentially augmented or transformed to operate successfully in different environments. Example use cases include generating correct-by-construction code augmented with checks that eliminate security vulnerabilities, automatically improving program comprehension and producing cleaner code, and automatically extracting the human knowledge in software and retargeting it to different languages and platforms.

We evaluate Konure and Shear by applying them to open source database-backed applications. Our results indicate that Konure and Shear work well with

applications whose core functionality is expressible in our DSLs, with data flow largely visible in the database queries and control flow directly tied to the query results. Our current techniques do not work well with applications whose core functionality is not expressible in our DSLs. Some of these applications can be inferred and regenerated after straightforward extensions to our current techniques. Others may require more involved extensions or are out of the scope of our approach.

Because our approach was designed to work with a range of inferrable computations, each of our techniques is unsuitable for inferring and regenerating general-purpose computations. To this end, our philosophy is to identify recurring patterns — many such patterns exist in today's software, which contains enormous amounts of human effort — and design domain-specific techniques that exploit these patterns. In the future, software will serve even more critical tasks in human society. The research presented in this thesis provides insights on how to reduce the fundamental inefficiencies in how people work with software.

Appendix A

Code Regenerated by KONURE

A.1 Regenerated Code for Fulcrum Task Manager

A.1.1 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)
```

A.1.2 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)
```

A.1.3 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)
```

A.1.4 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)
```

A.1.5 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)
```

A.1.6 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)
```

A.1.7 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)
```

A.1.8 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)
```

A.2 Regenerated Code for Kandan Chat Room

A.2.1 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)
```

A.2.2 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)
```

A.2.3 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)
```

A.2.4 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)
```

A.2.5 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)
```

A.2.6 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)
```

A.3 Regenerated Code for Enki Blogging Application

A.3.1 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)
```

A.3.2 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)
```

A.3.3 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)
```

A.3.4 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)
```

A.4 Regenerated Code for Blog Application

A.4.1 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)
```

A.4.2 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)
```

A.5 Regenerated Code for Student Registration System

A.5.1 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)
```

Appendix B

Synthetic Commands for Evaluating Konure

B.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]})
```

B.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", {})
```

B.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')})
```

B.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')})
```

B.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'))
```

Appendix C

Synthetic Commands for Evaluating SHEAR

C.1 Repetitions

C.1.1 Command repeat_2

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
```

C.1.2 Command repeat_3

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s3 = util.do_sql(conn, "SELECT * FROM t2", {})
```

C.1.3 Command repeat_4

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s3 = util.do_sql(conn, "SELECT * FROM t2", {})
    s4 = util.do_sql(conn, "SELECT * FROM t2", {})
```

C.1.4 Command `repeat_5`

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s3 = util.do_sql(conn, "SELECT * FROM t2", {})
    s4 = util.do_sql(conn, "SELECT * FROM t2", {})
    s5 = util.do_sql(conn, "SELECT * FROM t2", {})
```

C.2 Nested Loops

C.2.1 Command `nest`

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t0", {})
    for row0 in s0:
        s1 = util.do_sql(conn, "SELECT * FROM t1", {})
        for row1 in s1:
            s2 = util.do_sql(conn, "SELECT * FROM t2", {})
```

C.3 Consecutive Loops

C.3.1 Command `after_2`

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    for row0 in s0:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    for row2 in s2:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
```

C.3.2 Command after_3

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    for row0 in s0:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    for row2 in s2:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s4 = util.do_sql(conn, "SELECT * FROM t3", {})
    for row4 in s4:
        s5 = util.do_sql(conn, "SELECT * FROM t0", {})
```

C.3.3 Command after_4

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    for row0 in s0:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    for row2 in s2:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s4 = util.do_sql(conn, "SELECT * FROM t3", {})
    for row4 in s4:
        s5 = util.do_sql(conn, "SELECT * FROM t0", {})
    s6 = util.do_sql(conn, "SELECT * FROM t4", {})
```

```
    for row6 in s6:
        s7 = util.do_sql(conn, "SELECT * FROM t0", {})
```

C.3.4 Command after_5

```
with util.open_database('active_test_app') as conn:
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    for row0 in s0:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    for row2 in s2:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s4 = util.do_sql(conn, "SELECT * FROM t3", {})
    for row4 in s4:
        s5 = util.do_sql(conn, "SELECT * FROM t0", {})
    s6 = util.do_sql(conn, "SELECT * FROM t4", {})
    for row6 in s6:
        s7 = util.do_sql(conn, "SELECT * FROM t0", {})
    s8 = util.do_sql(conn, "SELECT * FROM t5", {})
    for row8 in s8:
        s9 = util.do_sql(conn, "SELECT * FROM t0", {})
```

C.4 Command example (Section 6.1)

```
with util.open_database('active_test_app') as conn:
    tasks1 = util.do_sql(conn, "SELECT * FROM tasks WHERE id = :x",
        {"x": sys.argv[1]})
    print(util.get_data(tasks1, 'tasks', 'title'))

    if util.has_rows(tasks1):
        task_assignee = util.get_one_data(tasks1, 'tasks', '
            assignee_id')
```

```
comments = util.do_sql(conn, "SELECT * FROM comments WHERE
   task_id = :x", {"x": sys.argv[1]})
print(util.get_data(comments, 'comments', 'content'))

for row2 in comments:
    comment_commenter = util.get_one_data(row2, 'comments',
        'commenter_id')
    users1 = util.do_sql(conn, "SELECT * FROM users WHERE
       id = :x", {"x": comment_commenter})
    print(util.get_data(users1, 'users', 'name'))
    tasks2 = util.do_sql(conn, "SELECT * FROM tasks WHERE
       creator_id = :x", {"x": comment_commenter})
    print(util.get_data(tasks2, 'tasks', 'title'))

users2 = util.do_sql(conn, "SELECT * FROM users WHERE id =
   :x", {"x": task_assignee})
print(util.get_data(users2, 'users', 'name'))
tasks3 = util.do_sql(conn, "SELECT * FROM tasks WHERE
   creator_id = :x", {"x": task_assignee})
print(util.get_data(tasks3, 'tasks', 'title'))
```

Appendix D

Code Regenerated by SHEAR

D.1 Regenerated Code for RailsCollab Project Manager

D.1.1 Command `get_projects_id_messages`

```
def get_projects_id_messages (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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
            INNER JOIN `people` ON `projects`.`id` = `people`.`
           project_id` WHERE `people`.`user_id` = :x0 AND (projects
           .completed_on IS NULL) ORDER BY projects.priority ASC,
           projects.name ASC", {'x0': util.get_one_data(s0, 'users'
           , 'id')})
        s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
           time_records`  WHERE `time_records`.`assigned_to_user_id
```

```
    ` = :x0 AND (start_date IS NOT NULL AND done_date IS
    NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
s4 = util.do_sql(conn, "SELECT  `projects`.* FROM `projects
    `  WHERE `projects`.`id` = :x0 LIMIT 1", {'x0': inputs
    [1]})
outputs.extend(util.get_data(s4, 'projects', 'id'))
outputs.extend(util.get_data(s4, 'projects', 'name'))
if util.has_rows(s4):
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
        companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
        'x0': util.get_one_data(s0, 'users', 'company_id')})
    outputs.extend(util.get_data(s5, 'companies', 'id'))
    outputs.extend(util.get_data(s5, 'companies', 'name'))
    outputs.extend(util.get_data(s5, 'companies', 'homepage
        '))
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
            `companies`  WHERE `companies`.`id` IS NULL
            LIMIT 1", {})
        s7 = util.do_sql(conn, "SELECT COUNT(*) FROM `
            messages`  WHERE `messages`.`project_id` = :x0",
             {'x0': inputs[1]})
        s8 = util.do_sql(conn, "SELECT COUNT(*) FROM `
            messages`  WHERE `messages`.`project_id` = :x0",
             {'x0': inputs[1]})
        if util.has_rows(s8):
            s18 = util.do_sql(conn, "SELECT  `messages`.*
                FROM `messages`  WHERE `messages`.`
                project_id` = :x0 ORDER BY created_on DESC
                LIMIT 10 OFFSET 0", {'x0': inputs[1]})
            outputs.extend(util.get_data(s18, 'messages', '
                id'))
            outputs.extend(util.get_data(s18, 'messages', '
                project_id'))
            outputs.extend(util.get_data(s18, 'messages', '
                title'))
```

```
outputs.extend(util.get_data(s18, 'messages', '
   text'))
s18_all = s18
for s18 in s18_all:
    s19 = util.do_sql(conn, "SELECT  `users`.*
       FROM `users`  WHERE `users`.`id` = :x0
       LIMIT 1", {'x0': util.get_one_data(s18,
       'messages', 'created_by_id')})
    outputs.extend(util.get_data(s19, 'users',
       'id'))
    outputs.extend(util.get_data(s19, 'users',
       'display_name'))
    s20 = util.do_sql(conn, "SELECT  `projects
       `.* FROM `projects`  WHERE `projects`.`
       id` = :x0 LIMIT 1", {'x0': util.
       get_one_data(s18, 'messages', '
       project_id')})
    outputs.extend(util.get_data(s20, 'projects
       ', 'id'))
    outputs.extend(util.get_data(s20, 'projects
       ', 'name'))
    s21 = util.do_sql(conn, "SELECT  `people`.*
        FROM `people`  WHERE (user_id = :x0 AND
        project_id = :x1) LIMIT 1", {'x0': util
       .get_one_data(s0, 'users', 'id'), 'x1':
       util.get_one_data(s18, 'messages', '
       project_id')})
    outputs.extend(util.get_data(s21, 'people',
        'project_id'))
    outputs.extend(util.get_data(s21, 'people',
        'user_id'))
    s22 = util.do_sql(conn, "SELECT `people`.*
       FROM `people`  WHERE `people`.`user_id`
       = :x0 AND `people`.`project_id` = :x1",
       {'x0': util.get_one_data(s0, 'users', '
       id'), 'x1': util.get_one_data(s18, '
```

```
            messages', 'project_id')})
        outputs.extend(util.get_data(s22, 'people',
            'project_id'))
        outputs.extend(util.get_data(s22, 'people',
            'user_id'))
        if util.has_rows(s22):
            pass
        else:
            s23 = util.do_sql(conn, "SELECT  `
                people`.* FROM `people`  WHERE (
                user_id = :x0 AND project_id = :x1)
                LIMIT 1", {'x0': util.get_one_data(
                s0, 'users', 'id'), 'x1': util.
                get_one_data(s18, 'messages', '
                project_id')})
            s24 = util.do_sql(conn, "SELECT `people
                `.* FROM `people`  WHERE `people`.`
                user_id` = :x0 AND `people`.`
                project_id` = :x1", {'x0': util.
                get_one_data(s0, 'users', 'id'), 'x1
                ': util.get_one_data(s18, 'messages'
                , 'project_id')})
    s18 = s18_all
    s25 = util.do_sql(conn, "SELECT  `people`.*
        FROM `people`  WHERE (user_id = :x0 AND
        project_id = :x1) LIMIT 1", {'x0': util.
        get_one_data(s0, 'users', 'id'), 'x1':
        inputs[1]})
    outputs.extend(util.get_data(s25, 'people', '
        project_id'))
    outputs.extend(util.get_data(s25, 'people', '
        user_id'))
    s26 = util.do_sql(conn, "SELECT COUNT(*) FROM `
        messages`  WHERE `messages`.`project_id` = :
        x0 AND `messages`.`is_important` = 1", {'x0'
        : inputs[1]})
```

```
s27 = util.do_sql(conn, "SELECT COUNT(*) FROM ‘
    categories‘  WHERE ‘categories‘.‘project_id‘
     = :x0", {’x0’: inputs[1]})
if util.has_rows(s27):
    s28 = util.do_sql(conn, "SELECT ‘categories
        ‘.* FROM ‘categories‘  WHERE ‘categories
        ‘.‘project_id‘ = :x0", {’x0’: inputs
        [1]})
    outputs.extend(util.get_data(s28, ’
        categories’, ’id’))
    outputs.extend(util.get_data(s28, ’
        categories’, ’project_id’))
    outputs.extend(util.get_data(s28, ’
        categories’, ’name’))
    s28_all = s28
    for s28 in s28_all:
        s29 = util.do_sql(conn, "SELECT  ‘
            projects‘.* FROM ‘projects‘  WHERE ‘
            projects‘.‘id‘ = :x0 LIMIT 1", {’x0’
            : util.get_one_data(s28, ’categories
            ’, ’project_id’)})
        outputs.extend(util.get_data(s29, ’
            projects’, ’id’))
        outputs.extend(util.get_data(s29, ’
            projects’, ’name’))
        s30 = util.do_sql(conn, "SELECT  ‘
            people‘.* FROM ‘people‘  WHERE (
            user_id = :x0 AND project_id = :x1)
            LIMIT 1", {’x0’: util.get_one_data(
            s0, ’users’, ’id’), ’x1’: util.
            get_one_data(s28, ’categories’, ’
            project_id’)})
        outputs.extend(util.get_data(s30, ’
            people’, ’project_id’))
        outputs.extend(util.get_data(s30, ’
            people’, ’user_id’))
```

```
                    s31 = util.do_sql(conn, "SELECT  `
                       people`.* FROM `people`  WHERE (
                       user_id = :x0 AND project_id = :x1)
                       LIMIT 1", {'x0': util.get_one_data(
                       s0, 'users', 'id'), 'x1': util.
                       get_one_data(s28, 'categories', '
                       project_id')})
                    outputs.extend(util.get_data(s31, '
                       people', 'project_id'))
                    outputs.extend(util.get_data(s31, '
                       people', 'user_id'))
                    if util.has_rows(s31):
                        pass
                    else:
                        s32 = util.do_sql(conn, "SELECT  `
                           people`.* FROM `people`  WHERE (
                           user_id = :x0 AND project_id = :
                           x1) LIMIT 1", {'x0': util.
                           get_one_data(s0, 'users', 'id'),
                            'x1': util.get_one_data(s28, '
                           categories', 'project_id')})
                        s33 = util.do_sql(conn, "SELECT  `
                           people`.* FROM `people`  WHERE (
                           user_id = :x0 AND project_id = :
                           x1) LIMIT 1", {'x0': util.
                           get_one_data(s0, 'users', 'id'),
                            'x1': util.get_one_data(s28, '
                           categories', 'project_id')})
            s28 = s28_all
            pass
        else:
            pass
    else:
        s9 = util.do_sql(conn, "SELECT  `people`.* FROM
            `people`  WHERE (user_id = :x0 AND
           project_id = :x1) LIMIT 1", {'x0': util.
```

```
    get_one_data(s0, 'users', 'id'), 'x1':
    inputs[1]})
outputs.extend(util.get_data(s9, 'people', '
    project_id'))
outputs.extend(util.get_data(s9, 'people', '
    user_id'))
s10 = util.do_sql(conn, "SELECT COUNT(*) FROM `
    messages`  WHERE `messages`.`project_id` = :
    x0 AND `messages`.`is_important` = 1", {'x0'
    : inputs[1]})
s11 = util.do_sql(conn, "SELECT COUNT(*) FROM `
    categories`  WHERE `categories`.`project_id`
     = :x0", {'x0': inputs[1]})
if util.has_rows(s11):
    s12 = util.do_sql(conn, "SELECT `categories
        `.* FROM `categories`  WHERE `categories
        `.`project_id` = :x0", {'x0': inputs
        [1]})
    outputs.extend(util.get_data(s12, '
        categories', 'id'))
    outputs.extend(util.get_data(s12, '
        categories', 'project_id'))
    outputs.extend(util.get_data(s12, '
        categories', 'name'))
    s12_all = s12
    for s12 in s12_all:
        s13 = util.do_sql(conn, "SELECT  `
            projects`.* FROM `projects`  WHERE `
            projects`.`id` = :x0 LIMIT 1", {'x0'
            : util.get_one_data(s12, 'categories
            ', 'project_id')})
        outputs.extend(util.get_data(s13, '
            projects', 'id'))
        outputs.extend(util.get_data(s13, '
            projects', 'name'))
```

```
s14 = util.do_sql(conn, "SELECT  `
   people`.* FROM `people`  WHERE (
   user_id = :x0 AND project_id = :x1)
   LIMIT 1", {'x0': util.get_one_data(
   s0, 'users', 'id'), 'x1': util.
   get_one_data(s12, 'categories', '
   project_id')})
outputs.extend(util.get_data(s14, '
   people', 'project_id'))
outputs.extend(util.get_data(s14, '
   people', 'user_id'))
s15 = util.do_sql(conn, "SELECT  `
   people`.* FROM `people`  WHERE (
   user_id = :x0 AND project_id = :x1)
   LIMIT 1", {'x0': util.get_one_data(
   s0, 'users', 'id'), 'x1': util.
   get_one_data(s12, 'categories', '
   project_id')})
outputs.extend(util.get_data(s15, '
   people', 'project_id'))
outputs.extend(util.get_data(s15, '
   people', 'user_id'))
if util.has_rows(s15):
    pass
else:
    s16 = util.do_sql(conn, "SELECT  `
       people`.* FROM `people`  WHERE (
       user_id = :x0 AND project_id = :
       x1) LIMIT 1", {'x0': util.
       get_one_data(s0, 'users', 'id'),
        'x1': util.get_one_data(s12, '
       categories', 'project_id')})
    s17 = util.do_sql(conn, "SELECT  `
       people`.* FROM `people`  WHERE (
       user_id = :x0 AND project_id = :
       x1) LIMIT 1", {'x0': util.
```

```
                                    get_one_data(s0, 'users', 'id'),
                                     'x1': util.get_one_data(s12, '
                                    categories', 'project_id')})
                        s12 = s12_all
                        pass
                    else:
                        pass
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.2 Command get_projects_id_messages_id

```
def get_projects_id_messages_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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
            INNER JOIN `people` ON `projects`.`id` = `people`.`
           project_id` WHERE `people`.`user_id` = :x0 AND (projects
           .completed_on IS NULL) ORDER BY projects.priority ASC,
           projects.name ASC", {'x0': util.get_one_data(s0, 'users'
           , 'id')})
        s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
           time_records`  WHERE `time_records`.`assigned_to_user_id
           ` = :x0 AND (start_date IS NOT NULL AND done_date IS
```

```
    NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
s4 = util.do_sql(conn, "SELECT  `projects`.* FROM `projects
    `  WHERE `projects`.`id` = :x0 LIMIT 1", {'x0': inputs
    [1]})
outputs.extend(util.get_data(s4, 'projects', 'id'))
if util.has_rows(s4):
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
        companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
        'x0': util.get_one_data(s0, 'users', 'company_id')})
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
            `companies`  WHERE `companies`.`id` IS NULL
            LIMIT 1", {})
        s7 = util.do_sql(conn, "SELECT  `messages`.* FROM `
            messages`  WHERE `messages`.`project_id` = :x0
            AND `messages`.`id` = :x1 ORDER BY created_on
            DESC LIMIT 1", {'x0': inputs[1], 'x1': inputs
            [2]})
        outputs.extend(util.get_data(s7, 'messages', 'id'))
        outputs.extend(util.get_data(s7, 'messages', '
            project_id'))
        outputs.extend(util.get_data(s7, 'messages', 'title
            '))
        outputs.extend(util.get_data(s7, 'messages', 'text'
            ))
        if util.has_rows(s7):
            s8 = util.do_sql(conn, "SELECT  `projects`.*
                FROM `projects`  WHERE `projects`.`id` = :x0
                 LIMIT 1", {'x0': inputs[1]})
            outputs.extend(util.get_data(s8, 'projects', '
                id'))
            outputs.extend(util.get_data(s8, 'projects', '
                name'))
            s9 = util.do_sql(conn, "SELECT  `people`.* FROM
                 `people`  WHERE (user_id = :x0 AND
                project_id = :x1) LIMIT 1", {'x0': util.
```

```
            get_one_data(s0, 'users', 'id'), 'x1':
            inputs[1]})
        outputs.extend(util.get_data(s9, 'people', '
            project_id'))
        outputs.extend(util.get_data(s9, 'people', '
            user_id'))
        if util.has_rows(s9):
            s11 = util.do_sql(conn, "SELECT  `users`.*
                FROM `users`  WHERE `users`.`id` = :x0
                LIMIT 1", {'x0': util.get_one_data(s7, '
                messages', 'created_by_id')})
            outputs.extend(util.get_data(s11, 'users',
                'id'))
            outputs.extend(util.get_data(s11, 'users',
                'display_name'))
            s12 = util.do_sql(conn, "SELECT  `
                milestones`.* FROM `milestones`  WHERE `
                milestones`.`id` = :x0 LIMIT 1", {'x0':
                util.get_one_data(s7, 'messages', '
                milestone_id')})
            outputs.extend(util.get_data(s12, '
                milestones', 'id'))
            outputs.extend(util.get_data(s12, '
                milestones', 'project_id'))
            outputs.extend(util.get_data(s12, '
                milestones', 'name'))
            s13 = util.do_sql(conn, "SELECT  `people`.*
                 FROM `people`  WHERE (user_id = :x0 AND
                 project_id = :x1) LIMIT 1", {'x0': util
                .get_one_data(s0, 'users', 'id'), 'x1':
                inputs[1]})
            outputs.extend(util.get_data(s13, 'people',
                 'project_id'))
            outputs.extend(util.get_data(s13, 'people',
                 'user_id'))
```

```
s14 = util.do_sql(conn, "SELECT  `people`.*
    FROM `people`  WHERE (user_id = :x0 AND
    project_id = :x1) LIMIT 1", {'x0': util
   .get_one_data(s0, 'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(s14, 'people',
    'project_id'))
outputs.extend(util.get_data(s14, 'people',
    'user_id'))
s15 = util.do_sql(conn, "SELECT  `people`.*
    FROM `people`  WHERE (user_id = :x0 AND
    project_id = :x1) LIMIT 1", {'x0': util
   .get_one_data(s0, 'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(s15, 'people',
    'project_id'))
outputs.extend(util.get_data(s15, 'people',
    'user_id'))
s16 = util.do_sql(conn, "SELECT  `people`.*
    FROM `people`  WHERE (user_id = :x0 AND
    project_id = :x1) LIMIT 1", {'x0': util
   .get_one_data(s0, 'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(s16, 'people',
    'project_id'))
outputs.extend(util.get_data(s16, 'people',
    'user_id'))
s17 = util.do_sql(conn, "SELECT  `
   categories`.* FROM `categories`  WHERE `
   categories`.`id` = :x0 LIMIT 1", {'x0':
   util.get_one_data(s7, 'messages', '
   category_id')})
outputs.extend(util.get_data(s17, '
   categories', 'id'))
outputs.extend(util.get_data(s17, '
   categories', 'name'))
```

```
            s18 = util.do_sql(conn, "SELECT `users`.*
                FROM `users` INNER JOIN `
                message_subscriptions` ON `users`.`id` =
                 `message_subscriptions`.`user_id` WHERE
                 `message_subscriptions`.`message_id` =
                :x0", {'x0': inputs[2]})
            outputs.extend(util.get_data(s18, 'users',
                'id'))
            outputs.extend(util.get_data(s18, 'users',
                'display_name'))
            s19 = util.do_sql(conn, "SELECT  `people`.*
                 FROM `people`  WHERE (user_id = :x0 AND
                 project_id = :x1) LIMIT 1", {'x0': util
                .get_one_data(s0, 'users', 'id'), 'x1':
                inputs[1]})
            outputs.extend(util.get_data(s19, 'people',
                 'project_id'))
            outputs.extend(util.get_data(s19, 'people',
                 'user_id'))
            s20 = util.do_sql(conn, "SELECT  `people`.*
                 FROM `people`  WHERE (user_id = :x0 AND
                 project_id = :x1) LIMIT 1", {'x0': util
                .get_one_data(s0, 'users', 'id'), 'x1':
                inputs[1]})
            outputs.extend(util.get_data(s20, 'people',
                 'project_id'))
            outputs.extend(util.get_data(s20, 'people',
                 'user_id'))
        else:
            s10 = util.do_sql(conn, "SELECT `people`.*
                FROM `people`  WHERE `people`.`user_id`
                = :x0 AND `people`.`project_id` = :x1",
                {'x0': util.get_one_data(s0, 'users', '
                id'), 'x1': inputs[1]})
    else:
        pass
```

```
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.3 Command get_projects_id_messages_display_list

```
def get_projects_id_messages_display_list (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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
            INNER JOIN `people` ON `projects`.`id` = `people`.`
           project_id` WHERE `people`.`user_id` = :x0 AND (projects
           .completed_on IS NULL) ORDER BY projects.priority ASC,
           projects.name ASC", {'x0': util.get_one_data(s0, 'users'
           , 'id')})
        s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
           time_records`  WHERE `time_records`.`assigned_to_user_id
           ` = :x0 AND (start_date IS NOT NULL AND done_date IS
           NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
        s4 = util.do_sql(conn, "SELECT  `projects`.* FROM `projects
           `  WHERE `projects`.`id` = :x0 LIMIT 1", {'x0': inputs
           [1]})
        outputs.extend(util.get_data(s4, 'projects', 'id'))
        outputs.extend(util.get_data(s4, 'projects', 'name'))
        if util.has_rows(s4):
```

```
s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
   companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
   'x0': util.get_one_data(s0, 'users', 'company_id')})
outputs.extend(util.get_data(s5, 'companies', 'id'))
outputs.extend(util.get_data(s5, 'companies', 'name'))
outputs.extend(util.get_data(s5, 'companies', 'homepage
   '))
if util.has_rows(s5):
    s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
       `companies`  WHERE `companies`.`id` IS NULL
       LIMIT 1", {})
    s7 = util.do_sql(conn, "SELECT COUNT(*) FROM `
       messages`  WHERE `messages`.`project_id` = :x0",
        {'x0': inputs[1]})
    s8 = util.do_sql(conn, "SELECT COUNT(*) FROM `
       messages`  WHERE `messages`.`project_id` = :x0",
        {'x0': inputs[1]})
    if util.has_rows(s8):
        s18 = util.do_sql(conn, "SELECT  `messages`.*
           FROM `messages`  WHERE `messages`.`
           project_id` = :x0 ORDER BY created_on DESC
           LIMIT 10 OFFSET 0", {'x0': inputs[1]})
        outputs.extend(util.get_data(s18, 'messages', '
           id'))
        outputs.extend(util.get_data(s18, 'messages', '
           project_id'))
        outputs.extend(util.get_data(s18, 'messages', '
           title'))
        s18_all = s18
        for s18 in s18_all:
            s19 = util.do_sql(conn, "SELECT  `projects
               `.* FROM `projects`  WHERE `projects`.`
               id` = :x0 LIMIT 1", {'x0': util.
               get_one_data(s18, 'messages', '
               project_id')})
```

```
outputs.extend(util.get_data(s19, 'projects
    ', 'id'))
outputs.extend(util.get_data(s19, 'projects
    ', 'name'))
s20 = util.do_sql(conn, "SELECT  `people`.*
     FROM `people`  WHERE (user_id = :x0 AND
     project_id = :x1) LIMIT 1", {'x0': util
    .get_one_data(s0, 'users', 'id'), 'x1':
    util.get_one_data(s18, 'messages', '
    project_id')})
outputs.extend(util.get_data(s20, 'people',
     'project_id'))
outputs.extend(util.get_data(s20, 'people',
     'user_id'))
s21 = util.do_sql(conn, "SELECT `people`.*
    FROM `people`  WHERE `people`.`user_id`
    = :x0 AND `people`.`project_id` = :x1",
    {'x0': util.get_one_data(s0, 'users', '
    id'), 'x1': util.get_one_data(s18, '
    messages', 'project_id')})
outputs.extend(util.get_data(s21, 'people',
     'project_id'))
outputs.extend(util.get_data(s21, 'people',
     'user_id'))
if util.has_rows(s21):
    s25 = util.do_sql(conn, "SELECT  `users
        `.* FROM `users`  WHERE `users`.`id`
         = :x0 LIMIT 1", {'x0': util.
        get_one_data(s18, 'messages', '
        created_by_id')})
    outputs.extend(util.get_data(s25, '
        users', 'id'))
    outputs.extend(util.get_data(s25, '
        users', 'display_name'))
else:
```

```
            s22 = util.do_sql(conn, "SELECT  `
               people`.* FROM `people`  WHERE (
               user_id = :x0 AND project_id = :x1)
               LIMIT 1", {'x0': util.get_one_data(
               s0, 'users', 'id'), 'x1': util.
               get_one_data(s18, 'messages', '
               project_id')})
            s23 = util.do_sql(conn, "SELECT `people
               `.* FROM `people`  WHERE `people`.`
               user_id` = :x0 AND `people`.`
               project_id` = :x1", {'x0': util.
               get_one_data(s0, 'users', 'id'), 'x1
               ': util.get_one_data(s18, 'messages'
               , 'project_id')})
            s24 = util.do_sql(conn, "SELECT  `users
               `.* FROM `users`  WHERE `users`.`id`
                = :x0 LIMIT 1", {'x0': util.
               get_one_data(s18, 'messages', '
               created_by_id')})
            outputs.extend(util.get_data(s24, '
               users', 'id'))
            outputs.extend(util.get_data(s24, '
               users', 'display_name'))
        s18 = s18_all
        s26 = util.do_sql(conn, "SELECT  `people`.*
           FROM `people`  WHERE (user_id = :x0 AND
           project_id = :x1) LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id'), 'x1':
           inputs[1]})
        outputs.extend(util.get_data(s26, 'people', '
           project_id'))
        outputs.extend(util.get_data(s26, 'people', '
           user_id'))
        s27 = util.do_sql(conn, "SELECT COUNT(*) FROM `
           messages`  WHERE `messages`.`project_id` = :
           x0 AND `messages`.`is_important` = 1", {'x0'
```

```
    : inputs[1]})
s28 = util.do_sql(conn, "SELECT COUNT(*) FROM ‘
    categories‘  WHERE ‘categories‘.‘project_id‘
     = :x0", {’x0’: inputs[1]})
if util.has_rows(s28):
    s29 = util.do_sql(conn, "SELECT ‘categories
        ‘.* FROM ‘categories‘  WHERE ‘categories
        ‘.‘project_id‘ = :x0", {’x0’: inputs
        [1]})
    outputs.extend(util.get_data(s29, ’
        categories’, ’id’))
    outputs.extend(util.get_data(s29, ’
        categories’, ’project_id’))
    outputs.extend(util.get_data(s29, ’
        categories’, ’name’))
    s29_all = s29
    for s29 in s29_all:
        s30 = util.do_sql(conn, "SELECT  ‘
            projects‘.* FROM ‘projects‘  WHERE ‘
            projects‘.‘id‘ = :x0 LIMIT 1", {’x0’
            : util.get_one_data(s29, ’categories
            ’, ’project_id’)})
        outputs.extend(util.get_data(s30, ’
            projects’, ’id’))
        outputs.extend(util.get_data(s30, ’
            projects’, ’name’))
        s31 = util.do_sql(conn, "SELECT  ‘
            people‘.* FROM ‘people‘  WHERE (
            user_id = :x0 AND project_id = :x1)
            LIMIT 1", {’x0’: util.get_one_data(
            s0, ’users’, ’id’), ’x1’: util.
            get_one_data(s29, ’categories’, ’
            project_id’)})
        outputs.extend(util.get_data(s31, ’
            people’, ’project_id’))
```

```
                    outputs.extend(util.get_data(s31, '
                       people', 'user_id'))
                    s32 = util.do_sql(conn, "SELECT  `
                       people`.* FROM `people`  WHERE (
                       user_id = :x0 AND project_id = :x1)
                       LIMIT 1", {'x0': util.get_one_data(
                       s0, 'users', 'id'), 'x1': util.
                       get_one_data(s29, 'categories', '
                       project_id')})
                    outputs.extend(util.get_data(s32, '
                       people', 'project_id'))
                    outputs.extend(util.get_data(s32, '
                       people', 'user_id'))
                    if util.has_rows(s32):
                        pass
                    else:
                        s33 = util.do_sql(conn, "SELECT  `
                           people`.* FROM `people`  WHERE (
                           user_id = :x0 AND project_id = :
                           x1) LIMIT 1", {'x0': util.
                           get_one_data(s0, 'users', 'id'),
                            'x1': util.get_one_data(s29, '
                           categories', 'project_id')})
                        s34 = util.do_sql(conn, "SELECT  `
                           people`.* FROM `people`  WHERE (
                           user_id = :x0 AND project_id = :
                           x1) LIMIT 1", {'x0': util.
                           get_one_data(s0, 'users', 'id'),
                            'x1': util.get_one_data(s29, '
                           categories', 'project_id')})
            s29 = s29_all
            pass
        else:
            pass
    else:
```

```
s9 = util.do_sql(conn, "SELECT  `people`.* FROM
    `people`  WHERE (user_id = :x0 AND
   project_id = :x1) LIMIT 1", {'x0': util.
   get_one_data(s0, 'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(s9, 'people', '
   project_id'))
outputs.extend(util.get_data(s9, 'people', '
   user_id'))
s10 = util.do_sql(conn, "SELECT COUNT(*) FROM `
   messages`  WHERE `messages`.`project_id` = :
   x0 AND `messages`.`is_important` = 1", {'x0'
   : inputs[1]})
s11 = util.do_sql(conn, "SELECT COUNT(*) FROM `
   categories`  WHERE `categories`.`project_id`
    = :x0", {'x0': inputs[1]})
if util.has_rows(s11):
    s12 = util.do_sql(conn, "SELECT `categories
       `.* FROM `categories`  WHERE `categories
       `.`project_id` = :x0", {'x0': inputs
       [1]})
    outputs.extend(util.get_data(s12, '
       categories', 'id'))
    outputs.extend(util.get_data(s12, '
       categories', 'project_id'))
    outputs.extend(util.get_data(s12, '
       categories', 'name'))
    s12_all = s12
    for s12 in s12_all:
        s13 = util.do_sql(conn, "SELECT  `
           projects`.* FROM `projects`  WHERE `
           projects`.`id` = :x0 LIMIT 1", {'x0'
           : util.get_one_data(s12, 'categories
           ', 'project_id')})
        outputs.extend(util.get_data(s13, '
           projects', 'id'))
```

```
outputs.extend(util.get_data(s13, '
   projects', 'name'))
s14 = util.do_sql(conn, "SELECT  `
   people`.* FROM `people`  WHERE (
   user_id = :x0 AND project_id = :x1)
   LIMIT 1", {'x0': util.get_one_data(
   s0, 'users', 'id'), 'x1': util.
   get_one_data(s12, 'categories', '
   project_id')})
outputs.extend(util.get_data(s14, '
   people', 'project_id'))
outputs.extend(util.get_data(s14, '
   people', 'user_id'))
s15 = util.do_sql(conn, "SELECT  `
   people`.* FROM `people`  WHERE (
   user_id = :x0 AND project_id = :x1)
   LIMIT 1", {'x0': util.get_one_data(
   s0, 'users', 'id'), 'x1': util.
   get_one_data(s12, 'categories', '
   project_id')})
outputs.extend(util.get_data(s15, '
   people', 'project_id'))
outputs.extend(util.get_data(s15, '
   people', 'user_id'))
if util.has_rows(s15):
    pass
else:
    s16 = util.do_sql(conn, "SELECT  `
       people`.* FROM `people`  WHERE (
       user_id = :x0 AND project_id = :
       x1) LIMIT 1", {'x0': util.
       get_one_data(s0, 'users', 'id'),
        'x1': util.get_one_data(s12, '
       categories', 'project_id')})
    s17 = util.do_sql(conn, "SELECT  `
       people`.* FROM `people`  WHERE (
```

```
                                    user_id = :x0 AND project_id = :
                                    x1) LIMIT 1", {'x0': util.
                                    get_one_data(s0, 'users', 'id'),
                                     'x1': util.get_one_data(s12, '
                                    categories', 'project_id')})
                        s12 = s12_all
                        pass
                    else:
                        pass
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.4 Command `get_projects_id_times`

For this command we use a modified version of RailsCollab, where we fixed a 500 error when a task's task list ID does not match any records in the database. Specifically, we changed the statement "`url_for hash_for_task_path(:id => self.id, :active_project => self.project_id, :only_path => host.nil?, :host => host)`" into "`url_for hash_for_task_path(:id => self.id, :active_project => self.project_id, :only_path => host.nil?, :host => host) rescue nil`" in the file `app/models/task.rb`.

```
def get_projects_id_times (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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
```

```
s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
    INNER JOIN `people` ON `projects`.`id` = `people`.`
   project_id` WHERE `people`.`user_id` = :x0 AND (projects
   .completed_on IS NULL) ORDER BY projects.priority ASC,
   projects.name ASC", {'x0': util.get_one_data(s0, 'users'
   , 'id')})
s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
   time_records`  WHERE `time_records`.`assigned_to_user_id
   ` = :x0 AND (start_date IS NOT NULL AND done_date IS
   NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
s4 = util.do_sql(conn, "SELECT  `projects`.* FROM `projects
   `  WHERE `projects`.`id` = :x0 LIMIT 1", {'x0': inputs
   [1]})
outputs.extend(util.get_data(s4, 'projects', 'id'))
outputs.extend(util.get_data(s4, 'projects', 'name'))
if util.has_rows(s4):
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
       companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
       'x0': util.get_one_data(s0, 'users', 'company_id')})
    outputs.extend(util.get_data(s5, 'companies', 'id'))
    outputs.extend(util.get_data(s5, 'companies', 'name'))
    outputs.extend(util.get_data(s5, 'companies', 'homepage
       '))
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
           `companies`  WHERE `companies`.`id` IS NULL
           LIMIT 1", {})
        s7 = util.do_sql(conn, "SELECT COUNT(*) FROM `
           time_records`  WHERE `time_records`.`project_id`
            = :x0", {'x0': inputs[1]})
        s8 = util.do_sql(conn, "SELECT COUNT(*) FROM `
           time_records`  WHERE `time_records`.`project_id`
            = :x0", {'x0': inputs[1]})
        if util.has_rows(s8):
            s10 = util.do_sql(conn, "SELECT  `time_records
               `.* FROM `time_records`  WHERE `time_records
```

```
    `.`project_id` = :x0 ORDER BY created_on
    DESC LIMIT 10 OFFSET 0", {'x0': inputs[1]})
outputs.extend(util.get_data(s10, 'time_records
    ', 'id'))
outputs.extend(util.get_data(s10, 'time_records
    ', 'project_id'))
outputs.extend(util.get_data(s10, 'time_records
    ', 'name'))
s10_all = s10
for s10 in s10_all:
    s11 = util.do_sql(conn, "SELECT  `companies
        `.* FROM `companies`  WHERE `companies
        `.`id` = :x0 LIMIT 1", {'x0': util.
        get_one_data(s10, 'time_records', '
        assigned_to_company_id')})
    outputs.extend(util.get_data(s11, '
        companies', 'name'))
    if util.has_rows(s11):
        s25 = util.do_sql(conn, "SELECT  `tasks
            `.* FROM `tasks`  WHERE `tasks`.`id`
             = :x0 LIMIT 1", {'x0': util.
            get_one_data(s10, 'time_records', '
            task_id')})
        if util.has_rows(s25):
            s31 = util.do_sql(conn, "SELECT  `
                task_lists`.* FROM `task_lists`
                 WHERE `task_lists`.`id` = :x0
                LIMIT 1", {'x0': util.
                get_one_data(s25, 'tasks', '
                task_list_id')})
            outputs.extend(util.get_data(s31, '
                task_lists', 'project_id'))
            s32 = util.do_sql(conn, "SELECT  `
                projects`.* FROM `projects`
                WHERE `projects`.`id` = :x0
                LIMIT 1", {'x0': util.
```

```
    get_one_data(s10, 'time_records'
    , 'project_id')})
outputs.extend(util.get_data(s32, '
    projects', 'id'))
outputs.extend(util.get_data(s32, '
    projects', 'name'))
s33 = util.do_sql(conn, "SELECT  `
    people`.* FROM `people`  WHERE (
    user_id = :x0 AND project_id = :
    x1) LIMIT 1", {'x0': util.
    get_one_data(s0, 'users', 'id'),
     'x1': util.get_one_data(s10, '
    time_records', 'project_id')})
outputs.extend(util.get_data(s33, '
    people', 'project_id'))
outputs.extend(util.get_data(s33, '
    people', 'user_id'))
s34 = util.do_sql(conn, "SELECT `
    people`.* FROM `people`  WHERE `
    people`.`user_id` = :x0 AND `
    people`.`project_id` = :x1", {'
    x0': util.get_one_data(s0, '
    users', 'id'), 'x1': util.
    get_one_data(s10, 'time_records'
    , 'project_id')})
outputs.extend(util.get_data(s34, '
    people', 'project_id'))
outputs.extend(util.get_data(s34, '
    people', 'user_id'))
if util.has_rows(s34):
    pass
else:
    s35 = util.do_sql(conn, "SELECT
         `people`.* FROM `people`
        WHERE (user_id = :x0 AND
        project_id = :x1) LIMIT 1",
```

```
                {'x0': util.get_one_data(s0,
                 'users', 'id'), 'x1': util.
                get_one_data(s10, '
                time_records', 'project_id')
                })
            s36 = util.do_sql(conn, "SELECT
                 `people`.* FROM `people`
                WHERE `people`.`user_id` = :
                x0 AND `people`.`project_id`
                 = :x1", {'x0': util.
                get_one_data(s0, 'users', '
                id'), 'x1': util.
                get_one_data(s10, '
                time_records', 'project_id')
                })
    else:
        s26 = util.do_sql(conn, "SELECT  `
            projects`.* FROM `projects`
            WHERE `projects`.`id` = :x0
            LIMIT 1", {'x0': util.
            get_one_data(s10, 'time_records'
            , 'project_id')})
        outputs.extend(util.get_data(s26, '
            projects', 'id'))
        outputs.extend(util.get_data(s26, '
            projects', 'name'))
        s27 = util.do_sql(conn, "SELECT  `
            people`.* FROM `people`  WHERE (
            user_id = :x0 AND project_id = :
            x1) LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id'),
             'x1': util.get_one_data(s10, '
            time_records', 'project_id')})
        outputs.extend(util.get_data(s27, '
            people', 'project_id'))
        outputs.extend(util.get_data(s27, '
```

```
                people', 'user_id'))
            s28 = util.do_sql(conn, "SELECT `
                people`.* FROM `people`  WHERE `
                people`.`user_id` = :x0 AND `
                people`.`project_id` = :x1", {'
                x0': util.get_one_data(s0, '
                users', 'id'), 'x1': util.
                get_one_data(s10, 'time_records'
                , 'project_id')})
            outputs.extend(util.get_data(s28, '
                people', 'project_id'))
            outputs.extend(util.get_data(s28, '
                people', 'user_id'))
            if util.has_rows(s28):
                pass
            else:
                s29 = util.do_sql(conn, "SELECT
                     `people`.* FROM `people`
                    WHERE (user_id = :x0 AND
                    project_id = :x1) LIMIT 1",
                    {'x0': util.get_one_data(s0,
                     'users', 'id'), 'x1': util.
                    get_one_data(s10, '
                    time_records', 'project_id')
                    })
                s30 = util.do_sql(conn, "SELECT
                     `people`.* FROM `people`
                    WHERE `people`.`user_id` = :
                    x0 AND `people`.`project_id`
                     = :x1", {'x0': util.
                    get_one_data(s0, 'users', '
                    id'), 'x1': util.
                    get_one_data(s10, '
                    time_records', 'project_id')
                    })
        else:
```

```
s12 = util.do_sql(conn, "SELECT  `users
   `.* FROM `users`  WHERE `users`.`id`
    = :x0 LIMIT 1", {'x0': util.
   get_one_data(s10, 'time_records', '
   assigned_to_user_id')})
outputs.extend(util.get_data(s12, '
   users', 'display_name'))
s13 = util.do_sql(conn, "SELECT  `tasks
   `.* FROM `tasks`  WHERE `tasks`.`id`
    = :x0 LIMIT 1", {'x0': util.
   get_one_data(s10, 'time_records', '
   task_id')})
if util.has_rows(s13):
    s19 = util.do_sql(conn, "SELECT  `
       task_lists`.* FROM `task_lists`
        WHERE `task_lists`.`id` = :x0
       LIMIT 1", {'x0': util.
       get_one_data(s13, 'tasks', '
       task_list_id')})
    outputs.extend(util.get_data(s19, '
       task_lists', 'project_id'))
    s20 = util.do_sql(conn, "SELECT  `
       projects`.* FROM `projects`
       WHERE `projects`.`id` = :x0
       LIMIT 1", {'x0': util.
       get_one_data(s10, 'time_records'
       , 'project_id')})
    outputs.extend(util.get_data(s20, '
       projects', 'id'))
    outputs.extend(util.get_data(s20, '
       projects', 'name'))
    s21 = util.do_sql(conn, "SELECT  `
       people`.* FROM `people`  WHERE (
       user_id = :x0 AND project_id = :
       x1) LIMIT 1", {'x0': util.
       get_one_data(s0, 'users', 'id'),
```

```
    'x1': util.get_one_data(s10, '
   time_records', 'project_id')})
outputs.extend(util.get_data(s21, '
   people', 'project_id'))
outputs.extend(util.get_data(s21, '
   people', 'user_id'))
s22 = util.do_sql(conn, "SELECT `
   people`.* FROM `people`  WHERE `
   people`.`user_id` = :x0 AND `
   people`.`project_id` = :x1", {'
   x0': util.get_one_data(s0, '
   users', 'id'), 'x1': util.
   get_one_data(s10, 'time_records'
   , 'project_id')})
outputs.extend(util.get_data(s22, '
   people', 'project_id'))
outputs.extend(util.get_data(s22, '
   people', 'user_id'))
if util.has_rows(s22):
    pass
else:
    s23 = util.do_sql(conn, "SELECT
         `people`.* FROM `people`
       WHERE (user_id = :x0 AND
       project_id = :x1) LIMIT 1",
       {'x0': util.get_one_data(s0,
        'users', 'id'), 'x1': util.
       get_one_data(s10, '
       time_records', 'project_id')
       })
    s24 = util.do_sql(conn, "SELECT
        `people`.* FROM `people`
       WHERE `people`.`user_id` = :
       x0 AND `people`.`project_id`
        = :x1", {'x0': util.
       get_one_data(s0, 'users', '
```

```
                id'), 'x1': util.
                get_one_data(s10, '
                time_records', 'project_id')
                })
        else:
            s14 = util.do_sql(conn, "SELECT  `
               projects`.* FROM `projects`
               WHERE `projects`.`id` = :x0
               LIMIT 1", {'x0': util.
               get_one_data(s10, 'time_records'
               , 'project_id')})
            outputs.extend(util.get_data(s14, '
               projects', 'id'))
            outputs.extend(util.get_data(s14, '
               projects', 'name'))
            s15 = util.do_sql(conn, "SELECT  `
               people`.* FROM `people`  WHERE (
               user_id = :x0 AND project_id = :
               x1) LIMIT 1", {'x0': util.
               get_one_data(s0, 'users', 'id'),
                'x1': util.get_one_data(s10, '
               time_records', 'project_id')})
            outputs.extend(util.get_data(s15, '
               people', 'project_id'))
            outputs.extend(util.get_data(s15, '
               people', 'user_id'))
            s16 = util.do_sql(conn, "SELECT `
               people`.* FROM `people`  WHERE `
               people`.`user_id` = :x0 AND `
               people`.`project_id` = :x1", {'
               x0': util.get_one_data(s0, '
               users', 'id'), 'x1': util.
               get_one_data(s10, 'time_records'
               , 'project_id')})
            outputs.extend(util.get_data(s16, '
               people', 'project_id'))
```

```
                        outputs.extend(util.get_data(s16, '
                           people', 'user_id'))
                        if util.has_rows(s16):
                            pass
                        else:
                            s17 = util.do_sql(conn, "SELECT
                                 `people`.* FROM `people`
                               WHERE (user_id = :x0 AND
                               project_id = :x1) LIMIT 1",
                               {'x0': util.get_one_data(s0,
                                'users', 'id'), 'x1': util.
                               get_one_data(s10, '
                               time_records', 'project_id')
                               })
                            s18 = util.do_sql(conn, "SELECT
                                 `people`.* FROM `people`
                               WHERE `people`.`user_id` = :
                               x0 AND `people`.`project_id`
                                = :x1", {'x0': util.
                               get_one_data(s0, 'users', '
                               id'), 'x1': util.
                               get_one_data(s10, '
                               time_records', 'project_id')
                               })
        s10 = s10_all
        s37 = util.do_sql(conn, "SELECT  `people`.*
           FROM `people`  WHERE (user_id = :x0 AND
           project_id = :x1) LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id'), 'x1':
           inputs[1]})
        outputs.extend(util.get_data(s37, 'people', '
           project_id'))
        outputs.extend(util.get_data(s37, 'people', '
           user_id'))
    else:
        s9 = util.do_sql(conn, "SELECT  `people`.* FROM
```

```
                        `people`  WHERE (user_id = :x0 AND
                        project_id = :x1) LIMIT 1", {'x0': util.
                        get_one_data(s0, 'users', 'id'), 'x1':
                        inputs[1]})
                    outputs.extend(util.get_data(s9, 'people', '
                        project_id'))
                    outputs.extend(util.get_data(s9, 'people', '
                        user_id'))
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.5 Command get_projects_id_times_id

```
def get_projects_id_times_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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
            WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
            get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
             INNER JOIN `people` ON `projects`.`id` = `people`.`
            project_id` WHERE `people`.`user_id` = :x0 AND (projects
            .completed_on IS NULL) ORDER BY projects.priority ASC,
            projects.name ASC", {'x0': util.get_one_data(s0, 'users'
            , 'id')})
        s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
            time_records`  WHERE `time_records`.`assigned_to_user_id
```

```
    ` = :x0 AND (start_date IS NOT NULL AND done_date IS
    NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
s4 = util.do_sql(conn, "SELECT  `projects`.* FROM `projects
    `  WHERE `projects`.`id` = :x0 LIMIT 1", {'x0': inputs
    [1]})
outputs.extend(util.get_data(s4, 'projects', 'id'))
if util.has_rows(s4):
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
        companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
        'x0': util.get_one_data(s0, 'users', 'company_id')})
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
            `companies`  WHERE `companies`.`id` IS NULL
            LIMIT 1", {})
        s7 = util.do_sql(conn, "SELECT  `time_records`.*
            FROM `time_records`  WHERE `time_records`.`
            project_id` = :x0 AND `time_records`.`id` = :x1
            LIMIT 1", {'x0': inputs[1], 'x1': inputs[2]})
        outputs.extend(util.get_data(s7, 'time_records', '
            id'))
        outputs.extend(util.get_data(s7, 'time_records', '
            project_id'))
        outputs.extend(util.get_data(s7, 'time_records', '
            name'))
        outputs.extend(util.get_data(s7, 'time_records', '
            description'))
        if util.has_rows(s7):
            s8 = util.do_sql(conn, "SELECT  `projects`.*
                FROM `projects`  WHERE `projects`.`id` = :x0
                 LIMIT 1", {'x0': inputs[1]})
            outputs.extend(util.get_data(s8, 'projects', '
                id'))
            outputs.extend(util.get_data(s8, 'projects', '
                name'))
            s9 = util.do_sql(conn, "SELECT  `people`.* FROM
                 `people`  WHERE (user_id = :x0 AND
```

```
    project_id = :x1) LIMIT 1", {'x0': util.
    get_one_data(s0, 'users', 'id'), 'x1':
    inputs[1]})
outputs.extend(util.get_data(s9, 'people', '
    project_id'))
outputs.extend(util.get_data(s9, 'people', '
    user_id'))
if util.has_rows(s9):
    s37 = util.do_sql(conn, "SELECT  `companies
        `.* FROM `companies`  WHERE `companies
        `.`id` = :x0 LIMIT 1", {'x0': util.
        get_one_data(s7, 'time_records', '
        assigned_to_company_id')})
    outputs.extend(util.get_data(s37, '
        companies', 'name'))
    if util.has_rows(s37):
        s47 = util.do_sql(conn, "SELECT  `tasks
            `.* FROM `tasks`  WHERE `tasks`.`id`
             = :x0 LIMIT 1", {'x0': util.
            get_one_data(s7, 'time_records', '
            task_id')})
        outputs.extend(util.get_data(s47, '
            tasks', 'id'))
        outputs.extend(util.get_data(s47, '
            tasks', 'text'))
        if util.has_rows(s47):
            s51 = util.do_sql(conn, "SELECT  `
                task_lists`.* FROM `task_lists`
                 WHERE `task_lists`.`id` = :x0
                LIMIT 1", {'x0': util.
                get_one_data(s47, 'tasks', '
                task_list_id')})
            outputs.extend(util.get_data(s51, '
                task_lists', 'project_id'))
            if util.has_rows(s51):
```

```
s52 = util.do_sql(conn, "SELECT
     `people`.* FROM `people`
   WHERE (user_id = :x0 AND
   project_id = :x1) LIMIT 1",
   {'x0': util.get_one_data(s0,
    'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(
   s52, 'people', 'project_id')
   )
outputs.extend(util.get_data(
   s52, 'people', 'user_id'))
s53 = util.do_sql(conn, "SELECT
     `people`.* FROM `people`
   WHERE (user_id = :x0 AND
   project_id = :x1) LIMIT 1",
   {'x0': util.get_one_data(s0,
    'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(
   s53, 'people', 'project_id')
   )
outputs.extend(util.get_data(
   s53, 'people', 'user_id'))
s54 = util.do_sql(conn, "SELECT
     `people`.* FROM `people`
   WHERE (user_id = :x0 AND
   project_id = :x1) LIMIT 1",
   {'x0': util.get_one_data(s0,
    'users', 'id'), 'x1':
   inputs[1]})
outputs.extend(util.get_data(
   s54, 'people', 'project_id')
   )
outputs.extend(util.get_data(
   s54, 'people', 'user_id'))
```

```
                    else:
                        pass
                else:
                    s48 = util.do_sql(conn, "SELECT  `
                        people`.* FROM `people`  WHERE (
                        user_id = :x0 AND project_id = :
                        x1) LIMIT 1", {'x0': util.
                        get_one_data(s0, 'users', 'id'),
                         'x1': inputs[1]})
                    outputs.extend(util.get_data(s48, '
                        people', 'project_id'))
                    outputs.extend(util.get_data(s48, '
                        people', 'user_id'))
                    s49 = util.do_sql(conn, "SELECT  `
                        people`.* FROM `people`  WHERE (
                        user_id = :x0 AND project_id = :
                        x1) LIMIT 1", {'x0': util.
                        get_one_data(s0, 'users', 'id'),
                         'x1': inputs[1]})
                    outputs.extend(util.get_data(s49, '
                        people', 'project_id'))
                    outputs.extend(util.get_data(s49, '
                        people', 'user_id'))
                    s50 = util.do_sql(conn, "SELECT  `
                        people`.* FROM `people`  WHERE (
                        user_id = :x0 AND project_id = :
                        x1) LIMIT 1", {'x0': util.
                        get_one_data(s0, 'users', 'id'),
                         'x1': inputs[1]})
                    outputs.extend(util.get_data(s50, '
                        people', 'project_id'))
                    outputs.extend(util.get_data(s50, '
                        people', 'user_id'))
            else:
                s38 = util.do_sql(conn, "SELECT  `users
                    `.* FROM `users`  WHERE `users`.`id`
```

```
    = :x0 LIMIT 1", {'x0': util.
   get_one_data(s7, 'time_records', '
   assigned_to_user_id')})
outputs.extend(util.get_data(s38, '
   users', 'display_name'))
s39 = util.do_sql(conn, "SELECT  `tasks
   `.* FROM `tasks`  WHERE `tasks`.`id`
    = :x0 LIMIT 1", {'x0': util.
   get_one_data(s7, 'time_records', '
   task_id')})
outputs.extend(util.get_data(s39, '
   tasks', 'id'))
outputs.extend(util.get_data(s39, '
   tasks', 'text'))
if util.has_rows(s39):
    s43 = util.do_sql(conn, "SELECT  `
       task_lists`.* FROM `task_lists`
        WHERE `task_lists`.`id` = :x0
       LIMIT 1", {'x0': util.
       get_one_data(s39, 'tasks', '
       task_list_id')})
    outputs.extend(util.get_data(s43, '
       task_lists', 'project_id'))
    if util.has_rows(s43):
        s44 = util.do_sql(conn, "SELECT
             `people`.* FROM `people`
           WHERE (user_id = :x0 AND
           project_id = :x1) LIMIT 1",
           {'x0': util.get_one_data(s0,
            'users', 'id'), 'x1':
           inputs[1]})
        outputs.extend(util.get_data(
           s44, 'people', 'project_id')
           )
        outputs.extend(util.get_data(
           s44, 'people', 'user_id'))
```

```
                s45 = util.do_sql(conn, "SELECT
                      `people`.* FROM `people`
                    WHERE (user_id = :x0 AND
                    project_id = :x1) LIMIT 1",
                    {'x0': util.get_one_data(s0,
                     'users', 'id'), 'x1':
                    inputs[1]})
                outputs.extend(util.get_data(
                    s45, 'people', 'project_id')
                    )
                outputs.extend(util.get_data(
                    s45, 'people', 'user_id'))
                s46 = util.do_sql(conn, "SELECT
                      `people`.* FROM `people`
                    WHERE (user_id = :x0 AND
                    project_id = :x1) LIMIT 1",
                    {'x0': util.get_one_data(s0,
                     'users', 'id'), 'x1':
                    inputs[1]})
                outputs.extend(util.get_data(
                    s46, 'people', 'project_id')
                    )
                outputs.extend(util.get_data(
                    s46, 'people', 'user_id'))
            else:
                pass
        else:
            s40 = util.do_sql(conn, "SELECT  `
                people`.* FROM `people`  WHERE (
                user_id = :x0 AND project_id = :
                x1) LIMIT 1", {'x0': util.
                get_one_data(s0, 'users', 'id'),
                 'x1': inputs[1]})
            outputs.extend(util.get_data(s40, '
                people', 'project_id'))
```

```
                outputs.extend(util.get_data(s40, '
                    people', 'user_id'))
                s41 = util.do_sql(conn, "SELECT  `
                    people`.* FROM `people`  WHERE (
                    user_id = :x0 AND project_id = :
                    x1) LIMIT 1", {'x0': util.
                    get_one_data(s0, 'users', 'id'),
                     'x1': inputs[1]})
                outputs.extend(util.get_data(s41, '
                    people', 'project_id'))
                outputs.extend(util.get_data(s41, '
                    people', 'user_id'))
                s42 = util.do_sql(conn, "SELECT  `
                    people`.* FROM `people`  WHERE (
                    user_id = :x0 AND project_id = :
                    x1) LIMIT 1", {'x0': util.
                    get_one_data(s0, 'users', 'id'),
                     'x1': inputs[1]})
                outputs.extend(util.get_data(s42, '
                    people', 'project_id'))
                outputs.extend(util.get_data(s42, '
                    people', 'user_id'))
        else:
            s10 = util.do_sql(conn, "SELECT  `people`.*
                 FROM `people`  WHERE (user_id = :x0 AND
                 project_id = :x1) LIMIT 1", {'x0': util
                .get_one_data(s0, 'users', 'id'), 'x1':
                inputs[1]})
            s11 = util.do_sql(conn, "SELECT  `companies
                `.* FROM `companies`  WHERE `companies
                `.`id` = :x0 LIMIT 1", {'x0': util.
                get_one_data(s7, 'time_records', '
                assigned_to_company_id')})
            outputs.extend(util.get_data(s11, '
                companies', 'name'))
            if util.has_rows(s11):
```

```
s25 = util.do_sql(conn, "SELECT  `tasks
   `.* FROM `tasks`  WHERE `tasks`.`id`
    = :x0 LIMIT 1", {'x0': util.
   get_one_data(s7, 'time_records', '
   task_id')})
outputs.extend(util.get_data(s25, '
   tasks', 'id'))
outputs.extend(util.get_data(s25, '
   tasks', 'text'))
if util.has_rows(s25):
    s31 = util.do_sql(conn, "SELECT  `
       task_lists`.* FROM `task_lists`
        WHERE `task_lists`.`id` = :x0
       LIMIT 1", {'x0': util.
       get_one_data(s25, 'tasks', '
       task_list_id')})
    outputs.extend(util.get_data(s31, '
       task_lists', 'project_id'))
    if util.has_rows(s31):
        s32 = util.do_sql(conn, "SELECT
             `people`.* FROM `people`
           WHERE (user_id = :x0 AND
           project_id = :x1) LIMIT 1",
           {'x0': util.get_one_data(s0,
            'users', 'id'), 'x1':
           inputs[1]})
        s33 = util.do_sql(conn, "SELECT
             `people`.* FROM `people`
           WHERE `people`.`user_id` = :
           x0 AND `people`.`project_id`
            = :x1", {'x0': util.
           get_one_data(s0, 'users', '
           id'), 'x1': inputs[1]})
        s34 = util.do_sql(conn, "SELECT
             `people`.* FROM `people`
           WHERE (user_id = :x0 AND
```

```
                    project_id = :x1) LIMIT 1",
                    {'x0': util.get_one_data(s0,
                     'users', 'id'), 'x1':
                    inputs[1]})
                s35 = util.do_sql(conn, "SELECT
                     `people`.* FROM `people`
                    WHERE `people`.`user_id` = :
                    x0 AND `people`.`project_id`
                     = :x1", {'x0': util.
                    get_one_data(s0, 'users', '
                    id'), 'x1': inputs[1]})
                s36 = util.do_sql(conn, "SELECT
                      `people`.* FROM `people`
                    WHERE (user_id = :x0 AND
                    project_id = :x1) LIMIT 1",
                    {'x0': util.get_one_data(s0,
                     'users', 'id'), 'x1':
                    inputs[1]})
            else:
                pass
        else:
            s26 = util.do_sql(conn, "SELECT  `
               people`.* FROM `people`  WHERE (
               user_id = :x0 AND project_id = :
               x1) LIMIT 1", {'x0': util.
               get_one_data(s0, 'users', 'id'),
                'x1': inputs[1]})
            s27 = util.do_sql(conn, "SELECT `
               people`.* FROM `people`  WHERE `
               people`.`user_id` = :x0 AND `
               people`.`project_id` = :x1", {'
               x0': util.get_one_data(s0, '
               users', 'id'), 'x1': inputs[1]})
            s28 = util.do_sql(conn, "SELECT  `
               people`.* FROM `people`  WHERE (
               user_id = :x0 AND project_id = :
```

```
                x1) LIMIT 1", {'x0': util.
                get_one_data(s0, 'users', 'id'),
                 'x1': inputs[1]})
            s29 = util.do_sql(conn, "SELECT `
                people`.* FROM `people`  WHERE `
                people`.`user_id` = :x0 AND `
                people`.`project_id` = :x1", {'
                x0': util.get_one_data(s0, '
                users', 'id'), 'x1': inputs[1]})
            s30 = util.do_sql(conn, "SELECT  `
                people`.* FROM `people`  WHERE (
                user_id = :x0 AND project_id = :
                x1) LIMIT 1", {'x0': util.
                get_one_data(s0, 'users', 'id'),
                 'x1': inputs[1]})
else:
    s12 = util.do_sql(conn, "SELECT  `users
        `.* FROM `users`  WHERE `users`.`id`
         = :x0 LIMIT 1", {'x0': util.
        get_one_data(s7, 'time_records', '
        assigned_to_user_id')})
    outputs.extend(util.get_data(s12, '
        users', 'display_name'))
    s13 = util.do_sql(conn, "SELECT  `tasks
        `.* FROM `tasks`  WHERE `tasks`.`id`
         = :x0 LIMIT 1", {'x0': util.
        get_one_data(s7, 'time_records', '
        task_id')})
    outputs.extend(util.get_data(s13, '
        tasks', 'id'))
    outputs.extend(util.get_data(s13, '
        tasks', 'text'))
    if util.has_rows(s13):
        s19 = util.do_sql(conn, "SELECT  `
            task_lists`.* FROM `task_lists`
             WHERE `task_lists`.`id` = :x0
```

```
    LIMIT 1", {'x0': util.
    get_one_data(s13, 'tasks', '
    task_list_id')})
outputs.extend(util.get_data(s19, '
    task_lists', 'project_id'))
if util.has_rows(s19):
    s20 = util.do_sql(conn, "SELECT
          `people`.* FROM `people`
        WHERE (user_id = :x0 AND
        project_id = :x1) LIMIT 1",
        {'x0': util.get_one_data(s0,
         'users', 'id'), 'x1':
        inputs[1]})
    s21 = util.do_sql(conn, "SELECT
         `people`.* FROM `people`
        WHERE `people`.`user_id` = :
        x0 AND `people`.`project_id`
         = :x1", {'x0': util.
        get_one_data(s0, 'users', '
        id'), 'x1': inputs[1]})
    s22 = util.do_sql(conn, "SELECT
          `people`.* FROM `people`
        WHERE (user_id = :x0 AND
        project_id = :x1) LIMIT 1",
        {'x0': util.get_one_data(s0,
         'users', 'id'), 'x1':
        inputs[1]})
    s23 = util.do_sql(conn, "SELECT
         `people`.* FROM `people`
        WHERE `people`.`user_id` = :
        x0 AND `people`.`project_id`
         = :x1", {'x0': util.
        get_one_data(s0, 'users', '
        id'), 'x1': inputs[1]})
    s24 = util.do_sql(conn, "SELECT
          `people`.* FROM `people`
```

```
                WHERE (user_id = :x0 AND
                project_id = :x1) LIMIT 1",
                {'x0': util.get_one_data(s0,
                 'users', 'id'), 'x1':
                inputs[1]})
        else:
            pass
    else:
        s14 = util.do_sql(conn, "SELECT  `
           people`.* FROM `people`  WHERE (
           user_id = :x0 AND project_id = :
           x1) LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id'),
            'x1': inputs[1]})
        s15 = util.do_sql(conn, "SELECT `
           people`.* FROM `people`  WHERE `
           people`.`user_id` = :x0 AND `
           people`.`project_id` = :x1", {'
           x0': util.get_one_data(s0, '
           users', 'id'), 'x1': inputs[1]})
        s16 = util.do_sql(conn, "SELECT  `
           people`.* FROM `people`  WHERE (
           user_id = :x0 AND project_id = :
           x1) LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id'),
            'x1': inputs[1]})
        s17 = util.do_sql(conn, "SELECT `
           people`.* FROM `people`  WHERE `
           people`.`user_id` = :x0 AND `
           people`.`project_id` = :x1", {'
           x0': util.get_one_data(s0, '
           users', 'id'), 'x1': inputs[1]})
        s18 = util.do_sql(conn, "SELECT  `
           people`.* FROM `people`  WHERE (
           user_id = :x0 AND project_id = :
           x1) LIMIT 1", {'x0': util.
```

```
                                    get_one_data(s0, 'users', 'id'),
                                     'x1': inputs[1]})
                else:
                    pass
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.6 Command get_projects_id_milestones_id

```
def get_projects_id_milestones_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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
            INNER JOIN `people` ON `projects`.`id` = `people`.`
           project_id` WHERE `people`.`user_id` = :x0 AND (projects
           .completed_on IS NULL) ORDER BY projects.priority ASC,
           projects.name ASC", {'x0': util.get_one_data(s0, 'users'
           , 'id')})
        s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
           time_records`  WHERE `time_records`.`assigned_to_user_id
           ` = :x0 AND (start_date IS NOT NULL AND done_date IS
           NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
        s4 = util.do_sql(conn, "SELECT  `projects`.* FROM `projects
           `  WHERE `projects`.`id` = :x0 LIMIT 1", {'x0': inputs
```

```
    [1]})
outputs.extend(util.get_data(s4, 'projects', 'id'))
if util.has_rows(s4):
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
       companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
       'x0': util.get_one_data(s0, 'users', 'company_id')})
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
           `companies`  WHERE `companies`.`id` IS NULL
           LIMIT 1", {})
        s7 = util.do_sql(conn, "SELECT  `milestones`.* FROM
            `milestones`  WHERE `milestones`.`project_id` =
            :x0 AND `milestones`.`id` = :x1 LIMIT 1", {'x0'
           : inputs[1], 'x1': inputs[2]})
        outputs.extend(util.get_data(s7, 'milestones', 'id'
           ))
        outputs.extend(util.get_data(s7, 'milestones', '
           project_id'))
        outputs.extend(util.get_data(s7, 'milestones', '
           name'))
        outputs.extend(util.get_data(s7, 'milestones', '
           description'))
        if util.has_rows(s7):
            s8 = util.do_sql(conn, "SELECT  `projects`.*
               FROM `projects`  WHERE `projects`.`id` = :x0
                LIMIT 1", {'x0': inputs[1]})
            outputs.extend(util.get_data(s8, 'projects', '
               id'))
            outputs.extend(util.get_data(s8, 'projects', '
               name'))
            s9 = util.do_sql(conn, "SELECT  `people`.* FROM
                `people`  WHERE (user_id = :x0 AND
               project_id = :x1) LIMIT 1", {'x0': util.
               get_one_data(s0, 'users', 'id'), 'x1':
               inputs[1]})
```

```
outputs.extend(util.get_data(s9, 'people', '
    project_id'))
outputs.extend(util.get_data(s9, 'people', '
    user_id'))
s10 = util.do_sql(conn, "SELECT `people`.* FROM
     `people`  WHERE `people`.`user_id` = :x0
    AND `people`.`project_id` = :x1", {'x0':
    util.get_one_data(s0, 'users', 'id'), 'x1':
    inputs[1]})
outputs.extend(util.get_data(s10, 'people', '
    project_id'))
outputs.extend(util.get_data(s10, 'people', '
    user_id'))
if util.has_rows(s10):
    s11 = util.do_sql(conn, "SELECT  `companies
        `.* FROM `companies`  WHERE `companies
        `.`id` = :x0 LIMIT 1", {'x0': util.
        get_one_data(s7, 'milestones', '
        assigned_to_company_id')})
    outputs.extend(util.get_data(s11, '
        companies', 'name'))
    if util.has_rows(s11):
        s19 = util.do_sql(conn, "SELECT `
            messages`.* FROM `messages`  WHERE `
            messages`.`milestone_id` = :x0", {'
            x0': inputs[2]})
        outputs.extend(util.get_data(s19, '
            messages', 'id'))
        outputs.extend(util.get_data(s19, '
            messages', 'milestone_id'))
        outputs.extend(util.get_data(s19, '
            messages', 'title'))
        s19_all = s19
        for s19 in s19_all:
            s20 = util.do_sql(conn, "SELECT  `
                users`.* FROM `users`  WHERE `
```

```
            users`.`id` = :x0 LIMIT 1", {'x0
            ': util.get_one_data(s19, '
            messages', 'created_by_id')})
        outputs.extend(util.get_data(s20, '
            users', 'id'))
        outputs.extend(util.get_data(s20, '
            users', 'display_name'))
    s19 = s19_all
    s21 = util.do_sql(conn, "SELECT `
        task_lists`.* FROM `task_lists`
        WHERE `task_lists`.`milestone_id` =
        :x0 ORDER BY `order` DESC", {'x0':
        inputs[2]})
    outputs.extend(util.get_data(s21, '
        task_lists', 'id'))
    outputs.extend(util.get_data(s21, '
        task_lists', 'milestone_id'))
    outputs.extend(util.get_data(s21, '
        task_lists', 'name'))
    s22 = util.do_sql(conn, "SELECT  `
        people`.* FROM `people`  WHERE (
        user_id = :x0 AND project_id = :x1)
        LIMIT 1", {'x0': util.get_one_data(
        s0, 'users', 'id'), 'x1': inputs
        [1]})
    outputs.extend(util.get_data(s22, '
        people', 'project_id'))
    outputs.extend(util.get_data(s22, '
        people', 'user_id'))
    s23 = util.do_sql(conn, "SELECT  `
        people`.* FROM `people`  WHERE (
        user_id = :x0 AND project_id = :x1)
        LIMIT 1", {'x0': util.get_one_data(
        s0, 'users', 'id'), 'x1': inputs
        [1]})
```

```
            outputs.extend(util.get_data(s23, '
               people', 'project_id'))
            outputs.extend(util.get_data(s23, '
               people', 'user_id'))
            s24 = util.do_sql(conn, "SELECT  `
               people`.* FROM `people`  WHERE (
               user_id = :x0 AND project_id = :x1)
               LIMIT 1", {'x0': util.get_one_data(
               s0, 'users', 'id'), 'x1': inputs
               [1]})
            outputs.extend(util.get_data(s24, '
               people', 'project_id'))
            outputs.extend(util.get_data(s24, '
               people', 'user_id'))
        else:
            s12 = util.do_sql(conn, "SELECT  `users
               `.* FROM `users`  WHERE `users`.`id`
                = :x0 LIMIT 1", {'x0': util.
               get_one_data(s7, 'milestones', '
               assigned_to_user_id')})
            outputs.extend(util.get_data(s12, '
               users', 'display_name'))
            s13 = util.do_sql(conn, "SELECT `
               messages`.* FROM `messages`  WHERE `
               messages`.`milestone_id` = :x0", {'
               x0': inputs[2]})
            outputs.extend(util.get_data(s13, '
               messages', 'id'))
            outputs.extend(util.get_data(s13, '
               messages', 'milestone_id'))
            outputs.extend(util.get_data(s13, '
               messages', 'title'))
            s13_all = s13
            for s13 in s13_all:
                s14 = util.do_sql(conn, "SELECT  `
                   users`.* FROM `users`  WHERE `
```

```
            users`.`id` = :x0 LIMIT 1", {'x0
            ': util.get_one_data(s13, '
            messages', 'created_by_id')})
        outputs.extend(util.get_data(s14, '
            users', 'id'))
        outputs.extend(util.get_data(s14, '
            users', 'display_name'))
    s13 = s13_all
    s15 = util.do_sql(conn, "SELECT `
        task_lists`.* FROM `task_lists`
        WHERE `task_lists`.`milestone_id` =
        :x0 ORDER BY `order` DESC", {'x0':
        inputs[2]})
    outputs.extend(util.get_data(s15, '
        task_lists', 'id'))
    outputs.extend(util.get_data(s15, '
        task_lists', 'milestone_id'))
    outputs.extend(util.get_data(s15, '
        task_lists', 'name'))
    s16 = util.do_sql(conn, "SELECT  `
        people`.* FROM `people`  WHERE (
        user_id = :x0 AND project_id = :x1)
        LIMIT 1", {'x0': util.get_one_data(
        s0, 'users', 'id'), 'x1': inputs
        [1]})
    outputs.extend(util.get_data(s16, '
        people', 'project_id'))
    outputs.extend(util.get_data(s16, '
        people', 'user_id'))
    s17 = util.do_sql(conn, "SELECT  `
        people`.* FROM `people`  WHERE (
        user_id = :x0 AND project_id = :x1)
        LIMIT 1", {'x0': util.get_one_data(
        s0, 'users', 'id'), 'x1': inputs
        [1]})
```

```
                            outputs.extend(util.get_data(s17, '
                                people', 'project_id'))
                            outputs.extend(util.get_data(s17, '
                                people', 'user_id'))
                            s18 = util.do_sql(conn, "SELECT  `
                                people`.* FROM `people`  WHERE (
                                user_id = :x0 AND project_id = :x1)
                                LIMIT 1", {'x0': util.get_one_data(
                                s0, 'users', 'id'), 'x1': inputs
                                [1]})
                            outputs.extend(util.get_data(s18, '
                                people', 'project_id'))
                            outputs.extend(util.get_data(s18, '
                                people', 'user_id'))
                    else:
                        pass
                else:
                    pass
            else:
                pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.7 Command get_projects

```
def get_projects (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', 'company_id'))
```

```
outputs.extend(util.get_data(s0, 'users', 'display_name'))
if util.has_rows(s0):
    s1 = 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(s1, 'users', 'id'))
    outputs.extend(util.get_data(s1, 'users', 'company_id'))
    outputs.extend(util.get_data(s1, 'users', 'display_name'))
    s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
        INNER JOIN `people` ON `projects`.`id` = `people`.`
       project_id` WHERE `people`.`user_id` = :x0 AND (projects
       .completed_on IS NULL) ORDER BY projects.priority ASC,
       projects.name ASC", {'x0': util.get_one_data(s0, 'users'
       , 'id')})
    s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
       time_records`  WHERE `time_records`.`assigned_to_user_id
       ` = :x0 AND (start_date IS NOT NULL AND done_date IS
       NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
    s4 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
       ", {})
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
       companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {'x0'
       : util.get_one_data(s0, 'users', 'company_id')})
    outputs.extend(util.get_data(s5, 'companies', 'id'))
    outputs.extend(util.get_data(s5, 'companies', 'name'))
    outputs.extend(util.get_data(s5, 'companies', 'homepage'))
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM `
           companies`  WHERE `companies`.`id` IS NULL LIMIT 1",
            {})
    else:
        pass
else:
    pass
return util.add_warnings(outputs)
```

D.1.8 Command get_companies_id

```
def get_companies_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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
            INNER JOIN `people` ON `projects`.`id` = `people`.`
           project_id` WHERE `people`.`user_id` = :x0 AND (projects
           .completed_on IS NULL) ORDER BY projects.priority ASC,
           projects.name ASC", {'x0': util.get_one_data(s0, 'users'
           , 'id')})
        s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
           time_records`  WHERE `time_records`.`assigned_to_user_id
           ` = :x0 AND (start_date IS NOT NULL AND done_date IS
           NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
        s4 = util.do_sql(conn, "SELECT  `companies`.* FROM `
           companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {'x0'
           : inputs[1]})
        outputs.extend(util.get_data(s4, 'companies', 'id'))
        outputs.extend(util.get_data(s4, 'companies', 'name'))
        outputs.extend(util.get_data(s4, 'companies', 'email'))
        outputs.extend(util.get_data(s4, 'companies', 'homepage'))
        outputs.extend(util.get_data(s4, 'companies', 'address'))
        outputs.extend(util.get_data(s4, 'companies', 'address2'))
        outputs.extend(util.get_data(s4, 'companies', 'city'))
        outputs.extend(util.get_data(s4, 'companies', 'state'))
        outputs.extend(util.get_data(s4, 'companies', 'zipcode'))
        outputs.extend(util.get_data(s4, 'companies', 'country'))
        outputs.extend(util.get_data(s4, 'companies', 'phone_number
           '))
```

```
        outputs.extend(util.get_data(s4, 'companies', 'fax_number')
           )
        if util.has_rows(s4):
            s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
               companies`  WHERE `companies`.`id` IS NULL LIMIT 1",
                {})
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.1.9 Command get_users_id

For this command we use a modified version of RailsCollab, where we fixed a 500 error when a user's IM type ID does not match any records in the database. Specifically, we changed the statement "`<td><img src="<%= im_value.im_type.icon_url %>" alt="<%= im_value.im_type.name %>" /></td>`" into "`<td><img src="<%= im_value.im_type.ic` `rescue nil %>" alt="<%= im_value.im_type.name rescue nil %>" /></td>`" in the file `app/views/users/_card.html.erb`.

```
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]})
    if util.has_rows(s0):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `projects`.* FROM `projects`
            INNER JOIN `people` ON `projects`.`id` = `people`.`
           project_id` WHERE `people`.`user_id` = :x0 AND (projects
           .completed_on IS NULL) ORDER BY projects.priority ASC,
           projects.name ASC", {'x0': util.get_one_data(s0, 'users'
```

```
    , 'id')})
s3 = util.do_sql(conn, "SELECT `time_records`.* FROM `
    time_records`  WHERE `time_records`.`assigned_to_user_id
    ` = :x0 AND (start_date IS NOT NULL AND done_date IS
    NULL)", {'x0': util.get_one_data(s0, 'users', 'id')})
s4 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
    WHERE `users`.`id` = :x0 LIMIT 1", {'x0': inputs[1]})
outputs.extend(util.get_data(s4, 'users', 'id'))
outputs.extend(util.get_data(s4, 'users', 'company_id'))
outputs.extend(util.get_data(s4, 'users', 'email'))
outputs.extend(util.get_data(s4, 'users', 'display_name'))
outputs.extend(util.get_data(s4, 'users', 'title'))
outputs.extend(util.get_data(s4, 'users', 'office_number'))
outputs.extend(util.get_data(s4, 'users', 'fax_number'))
outputs.extend(util.get_data(s4, 'users', 'mobile_number'))
outputs.extend(util.get_data(s4, 'users', 'home_number'))
if util.has_rows(s4):
    s5 = util.do_sql(conn, "SELECT  `companies`.* FROM `
        companies`  WHERE `companies`.`id` = :x0 LIMIT 1", {
        'x0': util.get_one_data(s0, 'users', 'company_id')})
    outputs.extend(util.get_data(s5, 'companies', 'id'))
    outputs.extend(util.get_data(s5, 'companies', 'name'))
    outputs.extend(util.get_data(s5, 'companies', 'homepage
        '))
    if util.has_rows(s5):
        s6 = util.do_sql(conn, "SELECT  `companies`.* FROM
            `companies`  WHERE `companies`.`id` IS NULL
            LIMIT 1", {})
        s7 = util.do_sql(conn, "SELECT  `companies`.* FROM
            `companies`  WHERE `companies`.`id` = :x0 LIMIT
            1", {'x0': util.get_one_data(s4, 'users', '
            company_id')})
        outputs.extend(util.get_data(s7, 'companies', 'id')
            )
        outputs.extend(util.get_data(s7, 'companies', 'name
            '))
```

```
        if util.has_rows(s7):
            s8 = util.do_sql(conn, "SELECT  `companies`.*
               FROM `companies`  WHERE `companies`.`id` IS
               NULL LIMIT 1", {})
            s9 = util.do_sql(conn, "SELECT COUNT(*) FROM `
               user_im_values`  WHERE `user_im_values`.`
               user_id` = :x0", {'x0': inputs[1]})
            if util.has_rows(s9):
                s10 = util.do_sql(conn, "SELECT `
                   user_im_values`.* FROM `user_im_values`
                    WHERE `user_im_values`.`user_id` = :x0
                   ORDER BY im_type_id DESC", {'x0': inputs
                   [1]})
                outputs.extend(util.get_data(s10, '
                   user_im_values', 'user_id'))
                outputs.extend(util.get_data(s10, '
                   user_im_values', 'value'))
                s10_all = s10
                for s10 in s10_all:
                    s11 = util.do_sql(conn, "SELECT  `
                       im_types`.* FROM `im_types`  WHERE `
                       im_types`.`id` = :x0 LIMIT 1", {'x0'
                       : util.get_one_data(s10, '
                       user_im_values', 'im_type_id')})
                    outputs.extend(util.get_data(s11, '
                       im_types', 'name'))
                    outputs.extend(util.get_data(s11, '
                       im_types', 'icon'))
                s10 = s10_all
                pass
            else:
                pass
        else:
            pass
    else:
        pass
```

```
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.2 Regenerated Code for Kanban Task Manager

D.2.1 Command get_api_lists

```
def get_api_lists (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):
        s1 = util.do_sql(conn, "SELECT `lists`.* FROM `lists` INNER
            JOIN `boards` ON `lists`.`board_id` = `boards`.`id`
           INNER JOIN `board_members` ON `boards`.`id` = `
           board_members`.`board_id` WHERE `board_members`.`
           member_id` = :x0  ORDER BY `lists`.`position` ASC", {'x0
           ': util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s1, 'lists', 'id'))
        outputs.extend(util.get_data(s1, 'lists', 'board_id'))
        outputs.extend(util.get_data(s1, 'lists', 'title'))
        outputs.extend(util.get_data(s1, 'lists', 'position'))
        s1_all = s1
        for s1 in s1_all:
            s2 = util.do_sql(conn, "SELECT  `boards`.* FROM `boards
               ` WHERE `boards`.`id` = :x0 LIMIT 1", {'x0': util.
               get_one_data(s1, 'lists', 'board_id')})
            outputs.extend(util.get_data(s2, 'boards', 'id'))
            s3 = util.do_sql(conn, "SELECT `cards`.* FROM `cards`
               WHERE `cards`.`list_id` = :x0  ORDER BY `cards`.`
```

```
    position` ASC", {'x0': util.get_one_data(s1, 'lists'
    , 'id')})
outputs.extend(util.get_data(s3, 'cards', 'id'))
outputs.extend(util.get_data(s3, 'cards', 'list_id'))
outputs.extend(util.get_data(s3, 'cards', 'title'))
outputs.extend(util.get_data(s3, 'cards', 'description'
    ))
outputs.extend(util.get_data(s3, 'cards', 'due_date'))
outputs.extend(util.get_data(s3, 'cards', 'position'))
s3_all = s3
for s3 in s3_all:
    s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `
        card_comments` WHERE `card_comments`.`card_id` =
         :x0", {'x0': util.get_one_data(s3, 'cards', 'id
        ')})
    s5 = util.do_sql(conn, "SELECT  `users`.* FROM `
        users` WHERE `users`.`id` = :x0  ORDER BY `users
        `.`id` ASC LIMIT 1", {'x0': util.get_one_data(s3
        , 'cards', 'assignee_id')})
    outputs.extend(util.get_data(s5, 'users', 'id'))
    outputs.extend(util.get_data(s5, 'users', 'email'))
    outputs.extend(util.get_data(s5, 'users', 'bio'))
    outputs.extend(util.get_data(s5, 'users', '
        full_name'))
    s6 = util.do_sql(conn, "SELECT `card_comments`.*
        FROM `card_comments` WHERE `card_comments`.`
        card_id` = :x0  ORDER BY `card_comments`.`
        created_at` DESC", {'x0': util.get_one_data(s3,
        'cards', 'id')})
    outputs.extend(util.get_data(s6, 'card_comments', '
        card_id'))
    outputs.extend(util.get_data(s6, 'card_comments', '
        content'))
    s6_all = s6
    for s6 in s6_all:
```

```
                s7 = util.do_sql(conn, "SELECT  `cards`.* FROM
                    `cards` WHERE `cards`.`id` = :x0  ORDER BY `
                    cards`.`position` ASC LIMIT 1", {'x0': util.
                    get_one_data(s6, 'card_comments', 'card_id')
                    })
                outputs.extend(util.get_data(s7, 'cards', 'id')
                    )
                outputs.extend(util.get_data(s7, 'cards', '
                    list_id'))
                outputs.extend(util.get_data(s7, 'cards', '
                    title'))
                outputs.extend(util.get_data(s7, 'cards', '
                    description'))
                outputs.extend(util.get_data(s7, 'cards', '
                    due_date'))
                outputs.extend(util.get_data(s7, 'cards', '
                    position'))
                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(s6, 'card_comments', '
                    commenter_id')})
                outputs.extend(util.get_data(s8, 'users', 'id')
                    )
                outputs.extend(util.get_data(s8, 'users', '
                    email'))
                outputs.extend(util.get_data(s8, 'users', 'bio'
                    ))
                outputs.extend(util.get_data(s8, 'users', '
                    full_name'))
            s6 = s6_all
            pass
        s3 = s3_all
        pass
    s1 = s1_all
    pass
```

```
    else:
        pass
    return util.add_warnings(outputs)
```

D.2.2 Command get_api_lists_id

```
def get_api_lists_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):
        s1 = util.do_sql(conn, "SELECT  `lists`.* FROM `lists`
           INNER JOIN `boards` ON `lists`.`board_id` = `boards`.`id
           ` INNER JOIN `board_members` ON `boards`.`id` = `
           board_members`.`board_id` WHERE `board_members`.`
           member_id` = :x0 AND `lists`.`id` = :x1  ORDER BY `lists
           `.`position` ASC LIMIT 1", {'x0': util.get_one_data(s0,
           'users', 'id'), 'x1': inputs[1]})
        outputs.extend(util.get_data(s1, 'lists', 'id'))
        outputs.extend(util.get_data(s1, 'lists', 'board_id'))
        outputs.extend(util.get_data(s1, 'lists', 'title'))
        outputs.extend(util.get_data(s1, 'lists', 'position'))
        if util.has_rows(s1):
            s2 = util.do_sql(conn, "SELECT  `boards`.* FROM `boards
               ` WHERE `boards`.`id` = :x0 LIMIT 1", {'x0': util.
               get_one_data(s1, 'lists', 'board_id')})
            outputs.extend(util.get_data(s2, 'boards', 'id'))
            s3 = util.do_sql(conn, "SELECT `cards`.* FROM `cards`
               WHERE `cards`.`list_id` = :x0  ORDER BY `cards`.`
               position` ASC", {'x0': inputs[1]})
            outputs.extend(util.get_data(s3, 'cards', 'id'))
            outputs.extend(util.get_data(s3, 'cards', 'list_id'))
            outputs.extend(util.get_data(s3, 'cards', 'title'))
```

```
outputs.extend(util.get_data(s3, 'cards', 'description'
    ))
outputs.extend(util.get_data(s3, 'cards', 'due_date'))
outputs.extend(util.get_data(s3, 'cards', 'position'))
s3_all = s3
for s3 in s3_all:
    s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `
        card_comments` WHERE `card_comments`.`card_id` =
         :x0", {'x0': util.get_one_data(s3, 'cards', 'id
        ')})
    s5 = util.do_sql(conn, "SELECT  `users`.* FROM `
        users` WHERE `users`.`id` = :x0  ORDER BY `users
        `.`id` ASC LIMIT 1", {'x0': util.get_one_data(s3
        , 'cards', 'assignee_id')})
    outputs.extend(util.get_data(s5, 'users', 'id'))
    outputs.extend(util.get_data(s5, 'users', 'email'))
    outputs.extend(util.get_data(s5, 'users', 'bio'))
    outputs.extend(util.get_data(s5, 'users', '
        full_name'))
    s6 = util.do_sql(conn, "SELECT `card_comments`.*
        FROM `card_comments` WHERE `card_comments`.`
        card_id` = :x0  ORDER BY `card_comments`.`
        created_at` DESC", {'x0': util.get_one_data(s3,
        'cards', 'id')})
    outputs.extend(util.get_data(s6, 'card_comments', '
        card_id'))
    outputs.extend(util.get_data(s6, 'card_comments', '
        content'))
    s6_all = s6
    for s6 in s6_all:
        s7 = util.do_sql(conn, "SELECT  `cards`.* FROM
            `cards` WHERE `cards`.`id` = :x0  ORDER BY `
            cards`.`position` ASC LIMIT 1", {'x0': util.
            get_one_data(s6, 'card_comments', 'card_id')
            })
```

```
                    outputs.extend(util.get_data(s7, 'cards', 'id')
                        )
                    outputs.extend(util.get_data(s7, 'cards', '
                        list_id'))
                    outputs.extend(util.get_data(s7, 'cards', '
                        title'))
                    outputs.extend(util.get_data(s7, 'cards', '
                        description'))
                    outputs.extend(util.get_data(s7, 'cards', '
                        due_date'))
                    outputs.extend(util.get_data(s7, 'cards', '
                        position'))
                    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(s6, 'card_comments', '
                        commenter_id')})
                    outputs.extend(util.get_data(s8, 'users', 'id')
                        )
                    outputs.extend(util.get_data(s8, 'users', '
                        email'))
                    outputs.extend(util.get_data(s8, 'users', 'bio'
                        ))
                    outputs.extend(util.get_data(s8, 'users', '
                        full_name'))
                s6 = s6_all
                pass
            s3 = s3_all
            pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.2.3 Command get_api_cards

```
def get_api_cards (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):
        s1 = util.do_sql(conn, "SELECT `cards`.* FROM `cards` INNER
            JOIN `lists` ON `cards`.`list_id` = `lists`.`id` INNER
           JOIN `boards` ON `lists`.`board_id` = `boards`.`id`
           INNER JOIN `board_members` ON `boards`.`id` = `
           board_members`.`board_id` WHERE `board_members`.`
           member_id` = :x0  ORDER BY `cards`.`position` ASC, `
           lists`.`position` ASC", {'x0': util.get_one_data(s0, '
           users', 'id')})
        outputs.extend(util.get_data(s1, 'cards', 'id'))
        outputs.extend(util.get_data(s1, 'cards', 'list_id'))
        outputs.extend(util.get_data(s1, 'cards', 'title'))
        outputs.extend(util.get_data(s1, 'cards', 'description'))
        outputs.extend(util.get_data(s1, 'cards', 'due_date'))
        outputs.extend(util.get_data(s1, 'cards', 'position'))
        s1_all = s1
        for s1 in s1_all:
            s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `
               card_comments` WHERE `card_comments`.`card_id` = :x0
               ", {'x0': util.get_one_data(s1, 'cards', 'id')})
            s3 = util.do_sql(conn, "SELECT  `lists`.* FROM `lists`
               WHERE `lists`.`id` = :x0  ORDER BY `lists`.`position
               ` ASC LIMIT 1", {'x0': util.get_one_data(s1, 'cards'
               , 'list_id')})
            outputs.extend(util.get_data(s3, 'lists', 'id'))
            s4 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
               WHERE `users`.`id` = :x0  ORDER BY `users`.`id` ASC
               LIMIT 1", {'x0': util.get_one_data(s1, 'cards', '
```

```
    assignee_id')})
outputs.extend(util.get_data(s4, 'users', 'id'))
outputs.extend(util.get_data(s4, 'users', 'email'))
outputs.extend(util.get_data(s4, 'users', 'bio'))
outputs.extend(util.get_data(s4, 'users', 'full_name'))
s5 = util.do_sql(conn, "SELECT `card_comments`.* FROM `
    card_comments` WHERE `card_comments`.`card_id` = :x0
      ORDER BY `card_comments`.`created_at` DESC", {'x0'
    : util.get_one_data(s1, 'cards', 'id')})
outputs.extend(util.get_data(s5, 'card_comments', '
    card_id'))
outputs.extend(util.get_data(s5, 'card_comments', '
    content'))
s5_all = s5
for s5 in s5_all:
    s6 = util.do_sql(conn, "SELECT  `cards`.* FROM `
        cards` WHERE `cards`.`id` = :x0  ORDER BY `cards
        `.`position` ASC LIMIT 1", {'x0': util.
        get_one_data(s5, 'card_comments', 'card_id')})
    outputs.extend(util.get_data(s6, 'cards', 'id'))
    outputs.extend(util.get_data(s6, 'cards', 'list_id'
        ))
    outputs.extend(util.get_data(s6, 'cards', 'title'))
    outputs.extend(util.get_data(s6, 'cards', '
        description'))
    outputs.extend(util.get_data(s6, 'cards', 'due_date
        '))
    outputs.extend(util.get_data(s6, 'cards', 'position
        '))
    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(s5
        , 'card_comments', 'commenter_id')})
    outputs.extend(util.get_data(s7, 'users', 'id'))
    outputs.extend(util.get_data(s7, 'users', 'email'))
    outputs.extend(util.get_data(s7, 'users', 'bio'))
```

```
                outputs.extend(util.get_data(s7, 'users', '
                    full_name'))
            s5 = s5_all
            pass
        s1 = s1_all
        pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.2.4 Command get_api_cards_id

```
def get_api_cards_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):
        s1 = util.do_sql(conn, "SELECT  `cards`.* FROM `cards`
            INNER JOIN `lists` ON `cards`.`list_id` = `lists`.`id`
            INNER JOIN `boards` ON `lists`.`board_id` = `boards`.`id
            ` INNER JOIN `board_members` ON `boards`.`id` = `
            board_members`.`board_id` WHERE `board_members`.`
            member_id` = :x0 AND `cards`.`id` = :x1  ORDER BY `cards
            `.`position` ASC, `lists`.`position` ASC LIMIT 1", {'x0'
            : util.get_one_data(s0, 'users', 'id'), 'x1': inputs
            [1]})
        outputs.extend(util.get_data(s1, 'cards', 'id'))
        outputs.extend(util.get_data(s1, 'cards', 'list_id'))
        outputs.extend(util.get_data(s1, 'cards', 'title'))
        outputs.extend(util.get_data(s1, 'cards', 'description'))
        outputs.extend(util.get_data(s1, 'cards', 'due_date'))
        outputs.extend(util.get_data(s1, 'cards', 'position'))
        if util.has_rows(s1):
```

```
s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `
   card_comments` WHERE `card_comments`.`card_id` = :x0
   ", {'x0': inputs[1]})
s3 = util.do_sql(conn, "SELECT  `lists`.* FROM `lists`
   WHERE `lists`.`id` = :x0  ORDER BY `lists`.`position
   ` ASC LIMIT 1", {'x0': util.get_one_data(s1, 'cards'
   , 'list_id')})
outputs.extend(util.get_data(s3, 'lists', 'id'))
s4 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
   WHERE `users`.`id` = :x0  ORDER BY `users`.`id` ASC
   LIMIT 1", {'x0': util.get_one_data(s1, 'cards', '
   assignee_id')})
outputs.extend(util.get_data(s4, 'users', 'id'))
outputs.extend(util.get_data(s4, 'users', 'email'))
outputs.extend(util.get_data(s4, 'users', 'bio'))
outputs.extend(util.get_data(s4, 'users', 'full_name'))
s5 = util.do_sql(conn, "SELECT `card_comments`.* FROM `
   card_comments` WHERE `card_comments`.`card_id` = :x0
     ORDER BY `card_comments`.`created_at` DESC", {'x0'
   : inputs[1]})
outputs.extend(util.get_data(s5, 'card_comments', '
   card_id'))
outputs.extend(util.get_data(s5, 'card_comments', '
   content'))
s5_all = s5
for s5 in s5_all:
    s6 = util.do_sql(conn, "SELECT  `cards`.* FROM `
       cards` WHERE `cards`.`id` = :x0  ORDER BY `cards
       `.`position` ASC LIMIT 1", {'x0': util.
       get_one_data(s5, 'card_comments', 'card_id')})
    outputs.extend(util.get_data(s6, 'cards', 'id'))
    outputs.extend(util.get_data(s6, 'cards', 'list_id'
       ))
    outputs.extend(util.get_data(s6, 'cards', 'title'))
    outputs.extend(util.get_data(s6, 'cards', '
       description'))
```

```
                outputs.extend(util.get_data(s6, 'cards', 'due_date
                    '))
                outputs.extend(util.get_data(s6, 'cards', 'position
                    '))
                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(s5
                    , 'card_comments', 'commenter_id')})
                outputs.extend(util.get_data(s7, 'users', 'id'))
                outputs.extend(util.get_data(s7, 'users', 'email'))
                outputs.extend(util.get_data(s7, 'users', 'bio'))
                outputs.extend(util.get_data(s7, 'users', '
                    full_name'))
            s5 = s5_all
            pass
        else:
            pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.2.5 Command get_api_boards_id

```
def get_api_boards_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):
        s1 = util.do_sql(conn, "SELECT  `boards`.* FROM `boards`
            INNER JOIN `board_members` ON `boards`.`id` = `
            board_members`.`board_id` WHERE `board_members`.`
            member_id` = :x0 AND `boards`.`id` = :x1 LIMIT 1", {'x0'
            : util.get_one_data(s0, 'users', 'id'), 'x1': inputs
```

```
    [1]})
outputs.extend(util.get_data(s1, 'boards', 'id'))
outputs.extend(util.get_data(s1, 'boards', 'name'))
outputs.extend(util.get_data(s1, 'boards', 'description'))
if util.has_rows(s1):
    s2 = util.do_sql(conn, "SELECT `lists`.* FROM `lists`
       WHERE `lists`.`board_id` = :x0  ORDER BY `lists`.`
       position` ASC", {'x0': inputs[1]})
    outputs.extend(util.get_data(s2, 'lists', 'id'))
    outputs.extend(util.get_data(s2, 'lists', 'board_id'))
    outputs.extend(util.get_data(s2, 'lists', 'title'))
    outputs.extend(util.get_data(s2, 'lists', 'position'))
    s2_all = s2
    for s2 in s2_all:
        s3 = util.do_sql(conn, "SELECT `cards`.* FROM `
           cards` WHERE `cards`.`list_id` = :x0  ORDER BY `
           cards`.`position` ASC", {'x0': util.get_one_data
           (s2, 'lists', 'id')})
        outputs.extend(util.get_data(s3, 'cards', 'id'))
        outputs.extend(util.get_data(s3, 'cards', 'list_id'
           ))
        outputs.extend(util.get_data(s3, 'cards', 'title'))
        outputs.extend(util.get_data(s3, 'cards', '
           description'))
        outputs.extend(util.get_data(s3, 'cards', 'due_date
           '))
        outputs.extend(util.get_data(s3, 'cards', 'position
           '))
        s3_all = s3
        for s3 in s3_all:
            s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `
               card_comments` WHERE `card_comments`.`
               card_id` = :x0", {'x0': util.get_one_data(s3
               , 'cards', 'id')})
            s5 = util.do_sql(conn, "SELECT  `users`.* FROM
               `users` WHERE `users`.`id` = :x0  ORDER BY `
```

```
    users`.`id` ASC LIMIT 1", {'x0': util.
    get_one_data(s3, 'cards', 'assignee_id')})
outputs.extend(util.get_data(s5, 'users', 'id')
    )
outputs.extend(util.get_data(s5, 'users', '
    email'))
outputs.extend(util.get_data(s5, 'users', 'bio'
    ))
outputs.extend(util.get_data(s5, 'users', '
    full_name'))
s6 = util.do_sql(conn, "SELECT `card_comments
    `.* FROM `card_comments` WHERE `
    card_comments`.`card_id` = :x0  ORDER BY `
    card_comments`.`created_at` DESC", {'x0':
    util.get_one_data(s3, 'cards', 'id')})
outputs.extend(util.get_data(s6, 'card_comments
    ', 'card_id'))
outputs.extend(util.get_data(s6, 'card_comments
    ', 'content'))
s6_all = s6
for s6 in s6_all:
    s7 = util.do_sql(conn, "SELECT  `cards`.*
        FROM `cards` WHERE `cards`.`id` = :x0
        ORDER BY `cards`.`position` ASC LIMIT 1"
        , {'x0': util.get_one_data(s6, '
        card_comments', 'card_id')})
    outputs.extend(util.get_data(s7, 'cards', '
        id'))
    outputs.extend(util.get_data(s7, 'cards', '
        list_id'))
    outputs.extend(util.get_data(s7, 'cards', '
        title'))
    outputs.extend(util.get_data(s7, 'cards', '
        description'))
    outputs.extend(util.get_data(s7, 'cards', '
        due_date'))
```

```
                    outputs.extend(util.get_data(s7, 'cards', '
                        position'))
                    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(s6, 'card_comments'
                        , 'commenter_id')})
                    outputs.extend(util.get_data(s8, 'users', '
                        id'))
                    outputs.extend(util.get_data(s8, 'users', '
                        email'))
                    outputs.extend(util.get_data(s8, 'users', '
                        bio'))
                    outputs.extend(util.get_data(s8, 'users', '
                        full_name'))
                s6 = s6_all
                pass
            s3 = s3_all
            pass
        s2 = s2_all
        s9 = util.do_sql(conn, "SELECT `users`.* FROM `users`
            INNER JOIN `board_members` ON `users`.`id` = `
            board_members`.`member_id` WHERE `board_members`.`
            board_id` = :x0  ORDER BY `users`.`id` ASC", {'x0':
            inputs[1]})
        outputs.extend(util.get_data(s9, 'users', 'id'))
        outputs.extend(util.get_data(s9, 'users', 'email'))
        outputs.extend(util.get_data(s9, 'users', 'bio'))
        outputs.extend(util.get_data(s9, 'users', 'full_name'))
    else:
        pass
else:
    pass
return util.add_warnings(outputs)
```

D.2.6 Command get_api_users_current

```
def get_api_users_current (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', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'bio'))
    outputs.extend(util.get_data(s0, 'users', 'full_name'))
    return util.add_warnings(outputs)
```

D.2.7 Command get_api_users_id

```
def get_api_users_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]})
    outputs.extend(util.get_data(s0, 'users', 'id'))
    outputs.extend(util.get_data(s0, 'users', 'email'))
    outputs.extend(util.get_data(s0, 'users', 'bio'))
    outputs.extend(util.get_data(s0, 'users', 'full_name'))
    return util.add_warnings(outputs)
```

D.3 Regenerated Code for Todo Task Manager

D.3.1 Command get_home

```
def get_home (conn, inputs):
    util.clear_warnings()
```

```
    outputs = []
    s0 = util.do_sql(conn, "SELECT `lists`.* FROM `lists`", {})
    outputs.extend(util.get_data(s0, 'lists', 'id'))
    outputs.extend(util.get_data(s0, 'lists', 'name'))
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT  1 AS one FROM `tasks` WHERE
            `tasks`.`list_id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'lists', 'id')})
        if util.has_rows(s1):
            s3 = util.do_sql(conn, "SELECT `tasks`.* FROM `tasks`
               WHERE `tasks`.`list_id` = :x0", {'x0': util.
               get_one_data(s0, 'lists', 'id')})
            outputs.extend(util.get_data(s3, 'tasks', 'id'))
            outputs.extend(util.get_data(s3, 'tasks', 'name'))
            outputs.extend(util.get_data(s3, 'tasks', 'list_id'))
            s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `tasks`
               WHERE `tasks`.`list_id` = :x0 AND `tasks`.`done` = 1
               ", {'x0': util.get_one_data(s0, 'lists', 'id')})
        else:
            s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `tasks`
               WHERE `tasks`.`list_id` = :x0 AND `tasks`.`done` = 1
               ", {'x0': util.get_one_data(s0, 'lists', 'id')})
    s0 = s0_all
    pass
    return util.add_warnings(outputs)
```

D.3.2 Command get_lists_id_tasks

```
def get_lists_id_tasks (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT `lists`.* FROM `lists`", {})
    s0_all = s0
    for s0 in s0_all:
```

```
        s1 = util.do_sql(conn, "SELECT  1 AS one FROM `tasks` WHERE
            `tasks`.`list_id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'lists', 'id')})
        if util.has_rows(s1):
            s3 = util.do_sql(conn, "SELECT `tasks`.* FROM `tasks`
               WHERE `tasks`.`list_id` = :x0", {'x0': util.
               get_one_data(s0, 'lists', 'id')})
            s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `tasks`
               WHERE `tasks`.`list_id` = :x0 AND `tasks`.`done` = 1
               ", {'x0': util.get_one_data(s0, 'lists', 'id')})
        else:
            s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `tasks`
               WHERE `tasks`.`list_id` = :x0 AND `tasks`.`done` = 1
               ", {'x0': util.get_one_data(s0, 'lists', 'id')})
    s0 = s0_all
    s5 = util.do_sql(conn, "SELECT  `lists`.* FROM `lists` WHERE `
       lists`.`id` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s5, 'lists', 'id'))
    outputs.extend(util.get_data(s5, 'lists', 'name'))
    return util.add_warnings(outputs)
```

D.3.3 Command `get_lists_id_tasks`

For this command we use a modified version of Todo, where we fixed a 404 error when the provided list ID does not match any records in the database. Specifically, we changed the statement "`prevlist = List.find listid`" into "`prevlist = List.find listid rescue nil`" in the file `app/views/tasks/index.html.erb`.

```
def get_lists_id_tasks (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT `lists`.* FROM `lists`", {})
    outputs.extend(util.get_data(s0, 'lists', 'id'))
    outputs.extend(util.get_data(s0, 'lists', 'name'))
    s0_all = s0
    for s0 in s0_all:
```

```
        s1 = util.do_sql(conn, "SELECT  1 AS one FROM `tasks` WHERE
            `tasks`.`list_id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'lists', 'id')})
        if util.has_rows(s1):
            s3 = util.do_sql(conn, "SELECT `tasks`.* FROM `tasks`
               WHERE `tasks`.`list_id` = :x0", {'x0': util.
               get_one_data(s0, 'lists', 'id')})
            outputs.extend(util.get_data(s3, 'tasks', 'id'))
            outputs.extend(util.get_data(s3, 'tasks', 'name'))
            outputs.extend(util.get_data(s3, 'tasks', 'list_id'))
            s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `tasks`
               WHERE `tasks`.`list_id` = :x0 AND `tasks`.`done` = 1
               ", {'x0': util.get_one_data(s0, 'lists', 'id')})
        else:
            s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `tasks`
               WHERE `tasks`.`list_id` = :x0 AND `tasks`.`done` = 1
               ", {'x0': util.get_one_data(s0, 'lists', 'id')})
    s0 = s0_all
    s5 = util.do_sql(conn, "SELECT  `lists`.* FROM `lists` WHERE `
       lists`.`id` = :x0 LIMIT 1", {'x0': inputs[0]})
    return util.add_warnings(outputs)
```

D.4 Regenerated Code for Fulcrum Task Manager

D.4.1 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):
```

```
        s1 = 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(s1, 'users', 'email'))
        s2 = 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(s0, 'users'
           , 'id')})
        outputs.extend(util.get_data(s2, 'projects', 'id'))
        outputs.extend(util.get_data(s2, 'projects', 'name'))
        outputs.extend(util.get_data(s2, 'projects', 'start_date'))
        s3 = 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(s3, 'users', 'email'))
        s4 = 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(s0, 'users'
           , 'id')})
        outputs.extend(util.get_data(s4, 'projects', 'id'))
        outputs.extend(util.get_data(s4, 'projects', 'name'))
        outputs.extend(util.get_data(s4, 'projects', 'start_date'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.2 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):
        s1 = 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(s1, 'users', 'email'))
        s2 = 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(s0, 'users'
           , 'id')})
        outputs.extend(util.get_data(s2, 'projects', 'id'))
        outputs.extend(util.get_data(s2, 'projects', 'name'))
        outputs.extend(util.get_data(s2, 'projects', 'start_date'))
        s3 = 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(s3, 'users', 'email'))
        s4 = 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(s0, 'users'
           , 'id')})
        outputs.extend(util.get_data(s4, 'projects', 'id'))
        outputs.extend(util.get_data(s4, 'projects', 'name'))
        outputs.extend(util.get_data(s4, 'projects', 'start_date'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.3 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):
    s1 = 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')})
    s2 = 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(s0, 'users'
       , 'id')})
    s3 = 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')})
    s4 = 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(s0, 'users', 'id'), 'x1': inputs
       [1]})
    outputs.extend(util.get_data(s4, 'projects', 'id'))
    outputs.extend(util.get_data(s4, 'projects', 'name'))
    if util.has_rows(s4):
        s6 = 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': inputs[1]})
        outputs.extend(util.get_data(s6, 'users', 'id'))
        outputs.extend(util.get_data(s6, 'users', 'email'))
        outputs.extend(util.get_data(s6, 'users', 'name'))
        outputs.extend(util.get_data(s6, 'users', 'initials'))
        s7 = 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(s0, 'users', 'id')})
            outputs.extend(util.get_data(s7, 'projects', 'id'))
            outputs.extend(util.get_data(s7, 'projects', 'name'))
        else:
            s5 = 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(s0, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.4 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):
        s1 = 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')})
        s2 = 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(s0, 'users'
           , 'id')})
        s3 = 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')})
```

```
s4 = 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(s0, 'users', 'id'), 'x1': inputs
    [1]})
if util.has_rows(s4):
    s6 = util.do_sql(conn, "SELECT `stories`.* FROM `
        stories`  WHERE `stories`.`project_id` IN (:x0)", {'
        x0': inputs[1]})
    outputs.extend(util.get_data(s6, 'stories', 'id'))
    outputs.extend(util.get_data(s6, 'stories', 'title'))
    outputs.extend(util.get_data(s6, 'stories', '
        description'))
    outputs.extend(util.get_data(s6, 'stories', 'estimate')
        )
    outputs.extend(util.get_data(s6, 'stories', '
        requested_by_id'))
    outputs.extend(util.get_data(s6, 'stories', '
        owned_by_id'))
    outputs.extend(util.get_data(s6, 'stories', 'project_id
        '))
    if util.has_rows(s6):
        s7 = util.do_sql(conn, "SELECT `notes`.* FROM `
            notes`  WHERE `notes`.`story_id` IN :x0", {'x0':
             util.get_data(s6, 'stories', 'id')})
        outputs.extend(util.get_data(s7, 'notes', 'id'))
        outputs.extend(util.get_data(s7, 'notes', 'note'))
        outputs.extend(util.get_data(s7, 'notes', 'user_id'
            ))
        outputs.extend(util.get_data(s7, 'notes', 'story_id
            '))
    else:
        pass
else:
```

```
            s5 = 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(s0, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.5 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):
        s1 = 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')})
        s2 = 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(s0, 'users'
           , 'id')})
        s3 = 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')})
        s4 = 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(s0, 'users', 'id'), 'x1': inputs
           [1]})
```

```
if util.has_rows(s4):
    s6 = util.do_sql(conn, "SELECT  `stories`.* FROM `
       stories`  WHERE `stories`.`project_id` = :x0 AND `
       stories`.`id` = :x1 LIMIT 1", {'x0': inputs[1], 'x1'
       : inputs[2]})
    outputs.extend(util.get_data(s6, 'stories', 'id'))
    outputs.extend(util.get_data(s6, 'stories', 'title'))
    outputs.extend(util.get_data(s6, 'stories', '
       description'))
    outputs.extend(util.get_data(s6, 'stories', 'estimate')
       )
    outputs.extend(util.get_data(s6, 'stories', '
       requested_by_id'))
    outputs.extend(util.get_data(s6, 'stories', '
       owned_by_id'))
    outputs.extend(util.get_data(s6, 'stories', 'project_id
       '))
    if util.has_rows(s6):
        s8 = util.do_sql(conn, "SELECT `notes`.* FROM `
           notes`  WHERE `notes`.`story_id` = :x0", {'x0':
           inputs[2]})
        outputs.extend(util.get_data(s8, 'notes', 'id'))
        outputs.extend(util.get_data(s8, 'notes', 'note'))
        outputs.extend(util.get_data(s8, 'notes', 'user_id'
           ))
        outputs.extend(util.get_data(s8, 'notes', 'story_id
           '))
    else:
        s7 = 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(s0, 'users', 'id')
           })
else:
```

```
            s5 = 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(s0, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.6 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):
        s1 = 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')})
        s2 = 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(s0, 'users'
           , 'id')})
        s3 = 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')})
        s4 = 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(s0, 'users', 'id'), 'x1': inputs
           [1]})
```

```
        if util.has_rows(s4):
            s6 = util.do_sql(conn, "SELECT  `stories`.* FROM `
               stories`  WHERE `stories`.`project_id` = :x0 AND `
               stories`.`id` = :x1 LIMIT 1", {'x0': inputs[1], 'x1'
               : inputs[2]})
            if util.has_rows(s6):
                s8 = util.do_sql(conn, "SELECT `notes`.* FROM `
                   notes`  WHERE `notes`.`story_id` = :x0", {'x0':
                   inputs[2]})
                outputs.extend(util.get_data(s8, 'notes', 'id'))
                outputs.extend(util.get_data(s8, 'notes', 'note'))
                outputs.extend(util.get_data(s8, 'notes', 'user_id'
                   ))
                outputs.extend(util.get_data(s8, 'notes', 'story_id
                   '))
            else:
                s7 = 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(s0, 'users', 'id')
                   })
        else:
            s5 = 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(s0, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.7 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):
    s1 = 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')})
    s2 = 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(s0, 'users'
       , 'id')})
    s3 = 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')})
    s4 = 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(s0, 'users', 'id'), 'x1': inputs
       [1]})
    if util.has_rows(s4):
        s6 = util.do_sql(conn, "SELECT  `stories`.* FROM `
           stories`  WHERE `stories`.`project_id` = :x0 AND `
           stories`.`id` = :x1 LIMIT 1", {'x0': inputs[1], 'x1'
           : inputs[2]})
        if util.has_rows(s6):
            s8 = util.do_sql(conn, "SELECT  `notes`.* FROM `
               notes`  WHERE `notes`.`story_id` = :x0 AND `
               notes`.`id` = :x1 LIMIT 1", {'x0': inputs[2], '
               x1': inputs[3]})
            outputs.extend(util.get_data(s8, 'notes', 'id'))
            outputs.extend(util.get_data(s8, 'notes', 'note'))
```

```
                outputs.extend(util.get_data(s8, 'notes', 'user_id'
                    ))
                outputs.extend(util.get_data(s8, 'notes', 'story_id
                    '))
                if util.has_rows(s8):
                    pass
                else:
                    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(s0, 'users', 'id')})
            else:
                s7 = 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(s0, 'users', 'id')
                    })
        else:
            s5 = 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(s0, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

D.4.8 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):
    s1 = 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')})
    s2 = 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(s0, 'users'
       , 'id')})
    s3 = 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')})
    s4 = 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(s0, 'users', 'id'), 'x1': inputs
       [1]})
    outputs.extend(util.get_data(s4, 'projects', 'id'))
    outputs.extend(util.get_data(s4, 'projects', 'name'))
    if util.has_rows(s4):
        s6 = 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': inputs[1]})
        outputs.extend(util.get_data(s6, 'users', 'id'))
        outputs.extend(util.get_data(s6, 'users', 'email'))
        outputs.extend(util.get_data(s6, 'users', 'name'))
        outputs.extend(util.get_data(s6, 'users', 'initials'))
        s7 = 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(s0, 'users', 'id')})
            outputs.extend(util.get_data(s7, 'projects', 'id'))
            outputs.extend(util.get_data(s7, 'projects', 'name'))
        else:
            s5 = 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(s0, 'users', 'id')})
    else:
        pass
    return util.add_warnings(outputs)
```

D.5 Regenerated Code for Kandan Chat Room

D.5.1 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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
           ", {})
        if util.has_rows(s2):
            s5 = util.do_sql(conn, "SELECT `activities`.* FROM `
               activities`  WHERE `activities`.`channel_id` IN :x0"
               , {'x0': util.get_data(s2, 'channels', 'id')})
            if util.has_rows(s5):
                s10 = util.do_sql(conn, "SELECT `users`.* FROM `
                   users`  WHERE `users`.`id` IN :x0", {'x0': util.
```

```
    get_data(s5, 'activities', 'user_id')})
s11 = util.do_sql(conn, "SELECT  `users`.* FROM `
    users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
    : util.get_one_data(s0, 'users', 'id')})
s12 = util.do_sql(conn, "SELECT `channels`.* FROM `
    channels`", {})
outputs.extend(util.get_data(s12, 'channels', 'id')
    )
outputs.extend(util.get_data(s12, 'channels', 'name
    '))
outputs.extend(util.get_data(s12, 'channels', '
    user_id'))
s12_all = s12
for s12 in s12_all:
    s13 = util.do_sql(conn, "SELECT COUNT(*) FROM `
        activities`  WHERE `activities`.`channel_id`
         = :x0", {'x0': util.get_one_data(s12, '
        channels', 'id')})
    s14 = 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(s12, '
        channels', 'id')})
    outputs.extend(util.get_data(s14, 'activities',
         'id'))
    outputs.extend(util.get_data(s14, 'activities',
         'content'))
    outputs.extend(util.get_data(s14, 'activities',
         'channel_id'))
    outputs.extend(util.get_data(s14, 'activities',
         'user_id'))
    if util.has_rows(s14):
        s15 = util.do_sql(conn, "SELECT `users`.*
            FROM `users`  WHERE `users`.`id` IN :x0"
            , {'x0': util.get_data(s14, 'activities'
            , 'user_id')})
```

```
                outputs.extend(util.get_data(s15, 'users',
                    'id'))
                outputs.extend(util.get_data(s15, 'users',
                    'email'))
                outputs.extend(util.get_data(s15, 'users',
                    'first_name'))
                outputs.extend(util.get_data(s15, 'users',
                    'last_name'))
                outputs.extend(util.get_data(s15, 'users',
                    'username'))
            else:
                pass
        s12 = s12_all
        pass
    else:
        s6 = util.do_sql(conn, "SELECT  `users`.* FROM `
            users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
            : util.get_one_data(s0, 'users', 'id')})
        s7 = util.do_sql(conn, "SELECT `channels`.* FROM `
            channels`", {})
        outputs.extend(util.get_data(s7, 'channels', 'id'))
        outputs.extend(util.get_data(s7, 'channels', 'name'
            ))
        outputs.extend(util.get_data(s7, 'channels', '
            user_id'))
        s7_all = s7
        for s7 in s7_all:
            s8 = util.do_sql(conn, "SELECT COUNT(*) FROM `
                activities`  WHERE `activities`.`channel_id`
                 = :x0", {'x0': util.get_one_data(s7, '
                channels', 'id')})
            s9 = 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(s7, '
                channels', 'id')})
```

```
                s7 = s7_all
                pass
        else:
            s3 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
                 WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
               get_one_data(s0, 'users', 'id')})
            s4 = util.do_sql(conn, "SELECT `channels`.* FROM `
               channels`", {})
    else:
        pass
    return util.add_warnings(outputs)
```

D.5.2 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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
           ", {})
        if util.has_rows(s2):
            s5 = util.do_sql(conn, "SELECT `activities`.* FROM `
               activities`  WHERE `activities`.`channel_id` IN :x0"
               , {'x0': util.get_data(s2, 'channels', 'id')})
            if util.has_rows(s5):
                s10 = util.do_sql(conn, "SELECT `users`.* FROM `
                   users`  WHERE `users`.`id` IN :x0", {'x0': util.
                   get_data(s5, 'activities', 'user_id')})
                s11 = util.do_sql(conn, "SELECT  `users`.* FROM `
                   users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
```

```
    : util.get_one_data(s0, 'users', 'id')})
s12 = util.do_sql(conn, "SELECT  `channels`.* FROM
    `channels`  WHERE `channels`.`id` = :x0 LIMIT 1"
    , {'x0': inputs[1]})
if util.has_rows(s12):
    s13 = util.do_sql(conn, "SELECT  `activities`.*
        FROM `activities`  WHERE `activities`.`
        channel_id` = :x0 ORDER BY id LIMIT 1", {'x0
        ': inputs[1]})
    outputs.extend(util.get_data(s13, 'activities',
        'id'))
    outputs.extend(util.get_data(s13, 'activities',
        'content'))
    outputs.extend(util.get_data(s13, 'activities',
        'channel_id'))
    outputs.extend(util.get_data(s13, 'activities',
        'user_id'))
    s14 = util.do_sql(conn, "SELECT  `activities`.*
        FROM `activities`  WHERE `activities`.`
        channel_id` = :x0 ORDER BY id DESC LIMIT 30"
        , {'x0': inputs[1]})
    outputs.extend(util.get_data(s14, 'activities',
        'id'))
    outputs.extend(util.get_data(s14, 'activities',
        'content'))
    outputs.extend(util.get_data(s14, 'activities',
        'channel_id'))
    outputs.extend(util.get_data(s14, 'activities',
        'user_id'))
    if util.has_rows(s14):
        s15 = util.do_sql(conn, "SELECT `users`.*
            FROM `users`  WHERE `users`.`id` IN :x0"
            , {'x0': util.get_data(s14, 'activities'
            , 'user_id')})
        outputs.extend(util.get_data(s15, 'users',
            'id'))
```

```
                outputs.extend(util.get_data(s15, 'users',
                    'email'))
                outputs.extend(util.get_data(s15, 'users',
                    'first_name'))
                outputs.extend(util.get_data(s15, 'users',
                    'last_name'))
                outputs.extend(util.get_data(s15, 'users',
                    'username'))
            else:
                pass
        else:
            pass
    else:
        s6 = util.do_sql(conn, "SELECT  `users`.* FROM `
           users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
           : util.get_one_data(s0, 'users', 'id')})
        s7 = util.do_sql(conn, "SELECT  `channels`.* FROM `
           channels`  WHERE `channels`.`id` = :x0 LIMIT 1",
            {'x0': inputs[1]})
        if util.has_rows(s7):
            s8 = util.do_sql(conn, "SELECT  `activities`.*
               FROM `activities`  WHERE `activities`.`
               channel_id` = :x0 ORDER BY id LIMIT 1", {'x0
               ': inputs[1]})
            s9 = util.do_sql(conn, "SELECT  `activities`.*
               FROM `activities`  WHERE `activities`.`
               channel_id` = :x0 ORDER BY id DESC LIMIT 30"
               , {'x0': inputs[1]})
        else:
            pass
else:
    s3 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
        WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
       get_one_data(s0, 'users', 'id')})
    s4 = util.do_sql(conn, "SELECT  `channels`.* FROM `
       channels`  WHERE `channels`.`id` = :x0 LIMIT 1", {'
```

```
                x0': inputs[1]})
    else:
        pass
    return util.add_warnings(outputs)
```

D.5.3 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):
        s1 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
           WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
           get_one_data(s0, 'users', 'id')})
        s2 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
           ", {})
        if util.has_rows(s2):
            s5 = util.do_sql(conn, "SELECT `activities`.* FROM `
               activities`  WHERE `activities`.`channel_id` IN :x0"
               , {'x0': util.get_data(s2, 'channels', 'id')})
            if util.has_rows(s5):
                s8 = util.do_sql(conn, "SELECT `users`.* FROM `
                   users`  WHERE `users`.`id` IN :x0", {'x0': util.
                   get_data(s5, 'activities', 'user_id')})
                s9 = util.do_sql(conn, "SELECT  `users`.* FROM `
                   users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
                   : util.get_one_data(s0, 'users', 'id')})
                s10 = util.do_sql(conn, "SELECT  `activities`.*
                   FROM `activities`  WHERE `activities`.`id` = :x0
                    LIMIT 1", {'x0': inputs[2]})
                outputs.extend(util.get_data(s10, 'activities', '
                   content'))
            else:
```

```
                s6 = util.do_sql(conn, "SELECT  `users`.* FROM `
                   users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
                   : util.get_one_data(s0, 'users', 'id')})
                s7 = util.do_sql(conn, "SELECT  `activities`.* FROM
                    `activities`  WHERE `activities`.`id` = :x0
                   LIMIT 1", {'x0': inputs[2]})
                outputs.extend(util.get_data(s7, 'activities', '
                   content'))
        else:
            s3 = util.do_sql(conn, "SELECT  `users`.* FROM `users`
                 WHERE `users`.`id` = :x0 LIMIT 1", {'x0': util.
               get_one_data(s0, 'users', 'id')})
            s4 = util.do_sql(conn, "SELECT  `activities`.* FROM `
               activities`  WHERE `activities`.`id` = :x0 LIMIT 1",
                {'x0': inputs[2]})
            outputs.extend(util.get_data(s4, 'activities', 'content
               '))
    else:
        pass
    return util.add_warnings(outputs)
```

D.5.4 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):
```

```
s1 = 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(s1, 'users', 'id'))
outputs.extend(util.get_data(s1, 'users', 'email'))
outputs.extend(util.get_data(s1, 'users', 'first_name'))
outputs.extend(util.get_data(s1, 'users', 'last_name'))
outputs.extend(util.get_data(s1, 'users', 'username'))
s2 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
   ", {})
if util.has_rows(s2):
    s4 = util.do_sql(conn, "SELECT `activities`.* FROM `
       activities`  WHERE `activities`.`channel_id` IN :x0"
       , {'x0': util.get_data(s2, 'channels', 'id')})
    if util.has_rows(s4):
        s6 = util.do_sql(conn, "SELECT `users`.* FROM `
           users`  WHERE `users`.`id` IN :x0", {'x0': util.
           get_data(s4, 'activities', 'user_id')})
        s7 = 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(s7, 'users', 'id'))
        outputs.extend(util.get_data(s7, 'users', 'email'))
        outputs.extend(util.get_data(s7, 'users', '
           first_name'))
        outputs.extend(util.get_data(s7, 'users', '
           last_name'))
        outputs.extend(util.get_data(s7, 'users', 'username
           '))
    else:
        s5 = 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(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:
            s3 = 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(s3, 'users', 'id'))
            outputs.extend(util.get_data(s3, 'users', 'email'))
            outputs.extend(util.get_data(s3, 'users', 'first_name')
                )
            outputs.extend(util.get_data(s3, 'users', 'last_name'))
            outputs.extend(util.get_data(s3, 'users', 'username'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.5.5 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):
        s1 = 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(s1, 'users', 'id'))
outputs.extend(util.get_data(s1, 'users', 'email'))
outputs.extend(util.get_data(s1, 'users', 'first_name'))
outputs.extend(util.get_data(s1, 'users', 'last_name'))
outputs.extend(util.get_data(s1, 'users', 'username'))
s2 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
    ", {})
if util.has_rows(s2):
    s5 = util.do_sql(conn, "SELECT `activities`.* FROM `
        activities`  WHERE `activities`.`channel_id` IN :x0"
        , {'x0': util.get_data(s2, 'channels', 'id')})
    if util.has_rows(s5):
        s8 = util.do_sql(conn, "SELECT `users`.* FROM `
            users`  WHERE `users`.`id` IN :x0", {'x0': util.
            get_data(s5, 'activities', 'user_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  `users`.* FROM `
            users`  WHERE `users`.`id` = :x0 LIMIT 1", {'x0'
            : util.get_one_data(s0, 'users', 'id')})
        outputs.extend(util.get_data(s9, 'users', 'id'))
        outputs.extend(util.get_data(s9, 'users', 'email'))
        outputs.extend(util.get_data(s9, 'users', '
            first_name'))
        outputs.extend(util.get_data(s9, 'users', '
            last_name'))
        outputs.extend(util.get_data(s9, 'users', 'username
            '))
```

```
            s10 = util.do_sql(conn, "SELECT `users`.* FROM `
                users`", {})
            outputs.extend(util.get_data(s10, 'users', 'id'))
            outputs.extend(util.get_data(s10, 'users', 'email')
                )
            outputs.extend(util.get_data(s10, 'users', '
                first_name'))
            outputs.extend(util.get_data(s10, 'users', '
                last_name'))
            outputs.extend(util.get_data(s10, 'users', '
                username'))
        else:
            s6 = 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(s6, 'users', 'id'))
            outputs.extend(util.get_data(s6, 'users', 'email'))
            outputs.extend(util.get_data(s6, 'users', '
                first_name'))
            outputs.extend(util.get_data(s6, 'users', '
                last_name'))
            outputs.extend(util.get_data(s6, 'users', 'username
                '))
            s7 = util.do_sql(conn, "SELECT `users`.* FROM `
                users`", {})
            outputs.extend(util.get_data(s7, 'users', 'id'))
            outputs.extend(util.get_data(s7, 'users', 'email'))
            outputs.extend(util.get_data(s7, 'users', '
                first_name'))
            outputs.extend(util.get_data(s7, 'users', '
                last_name'))
            outputs.extend(util.get_data(s7, 'users', 'username
                '))
    else:
        s3 = 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(s3, 'users', 'id'))
            outputs.extend(util.get_data(s3, 'users', 'email'))
            outputs.extend(util.get_data(s3, 'users', 'first_name')
                )
            outputs.extend(util.get_data(s3, 'users', 'last_name'))
            outputs.extend(util.get_data(s3, 'users', 'username'))
            s4 = util.do_sql(conn, "SELECT `users`.* FROM `users`",
                 {})
            outputs.extend(util.get_data(s4, 'users', 'id'))
            outputs.extend(util.get_data(s4, 'users', 'email'))
            outputs.extend(util.get_data(s4, 'users', 'first_name')
                )
            outputs.extend(util.get_data(s4, 'users', 'last_name'))
            outputs.extend(util.get_data(s4, 'users', 'username'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.5.6 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):
        s1 = 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(s1, 'users', 'id'))
outputs.extend(util.get_data(s1, 'users', 'email'))
outputs.extend(util.get_data(s1, 'users', 'first_name'))
outputs.extend(util.get_data(s1, 'users', 'last_name'))
outputs.extend(util.get_data(s1, 'users', 'username'))
s2 = util.do_sql(conn, "SELECT `channels`.* FROM `channels`
   ", {})
if util.has_rows(s2):
    s4 = util.do_sql(conn, "SELECT `activities`.* FROM `
       activities`  WHERE `activities`.`channel_id` IN :x0"
       , {'x0': util.get_data(s2, 'channels', 'id')})
    if util.has_rows(s4):
        s6 = util.do_sql(conn, "SELECT `users`.* FROM `
           users`  WHERE `users`.`id` IN :x0", {'x0': util.
           get_data(s4, 'activities', 'user_id')})
        s7 = 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(s7, 'users', 'id'))
        outputs.extend(util.get_data(s7, 'users', 'email'))
        outputs.extend(util.get_data(s7, 'users', '
           first_name'))
        outputs.extend(util.get_data(s7, 'users', '
           last_name'))
        outputs.extend(util.get_data(s7, 'users', 'username
           '))
    else:
        s5 = 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(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:
            s3 = 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(s3, 'users', 'id'))
            outputs.extend(util.get_data(s3, 'users', 'email'))
            outputs.extend(util.get_data(s3, 'users', 'first_name')
                )
            outputs.extend(util.get_data(s3, 'users', 'last_name'))
            outputs.extend(util.get_data(s3, 'users', 'username'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.6 Regenerated Code for Enki Blogging Application

D.6.1 Command `get_home`

For this command we use the original version of Enki, but disregard all queries on the `taggings` table.

```
def get_home (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT COUNT(count_column) FROM (SELECT
         1 AS count_column FROM `posts` WHERE (1=1) LIMIT 15)
        subquery_for_count", {})
    if util.has_rows(s0):
        s4 = util.do_sql(conn, "SELECT  `posts`.* FROM `posts`
            WHERE (1=1)  ORDER BY published_at DESC LIMIT 15", {})
        outputs.extend(util.get_data(s4, 'posts', 'id'))
        outputs.extend(util.get_data(s4, 'posts', 'title'))
```

```
        outputs.extend(util.get_data(s4, 'posts', 'slug'))
        outputs.extend(util.get_data(s4, 'posts', 'body_html'))
        s4_all = s4
        for s4 in s4_all:
            s5 = util.do_sql(conn, "SELECT COUNT(*) FROM `comments`
                 WHERE `comments`.`post_id` = :x0", {'x0': util.
                get_one_data(s4, 'posts', 'id')})
        s4 = s4_all
        s6 = util.do_sql(conn, "SELECT `pages`.* FROM `pages`
           ORDER BY title", {})
        outputs.extend(util.get_data(s6, 'pages', 'title'))
        outputs.extend(util.get_data(s6, 'pages', 'slug'))
        s7 = util.do_sql(conn, "SELECT `posts`.* FROM `posts` WHERE
            (`posts`.`published_at` IS NOT NULL)", {})
        s8 = util.do_sql(conn, "SELECT `tags`.* FROM `tags` WHERE `
           tags`.`name` IS NULL", {})
    else:
        s1 = util.do_sql(conn, "SELECT `pages`.* FROM `pages`
           ORDER BY title", {})
        outputs.extend(util.get_data(s1, 'pages', 'title'))
        outputs.extend(util.get_data(s1, 'pages', 'slug'))
        s2 = util.do_sql(conn, "SELECT `posts`.* FROM `posts` WHERE
            (`posts`.`published_at` IS NOT NULL)", {})
        s3 = util.do_sql(conn, "SELECT `tags`.* FROM `tags` WHERE
           1=0", {})
    return util.add_warnings(outputs)
```

D.6.2 Command get_archives

```
def get_archives (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT `posts`.* FROM `posts` WHERE
       (1=1)  ORDER BY published_at DESC", {})
    outputs.extend(util.get_data(s0, 'posts', 'title'))
```

```
    outputs.extend(util.get_data(s0, 'posts', 'slug'))
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT  1 AS one FROM `tags` INNER
           JOIN `taggings` ON `tags`.`id` = `taggings`.`tag_id`
           WHERE `taggings`.`taggable_id` = :x0 AND `taggings`.`
           taggable_type` = 'Post' AND `taggings`.`context` = 'tags
           ' LIMIT 1", {'x0': util.get_one_data(s0, 'posts', 'id')
           })
        if util.has_rows(s1):
            s2 = util.do_sql(conn, "SELECT `tags`.* FROM `tags`
               INNER JOIN `taggings` ON `tags`.`id` = `taggings`.`
               tag_id` WHERE `taggings`.`taggable_id` = :x0 AND `
               taggings`.`taggable_type` = 'Post' AND `taggings`.`
               context` = 'tags'", {'x0': util.get_one_data(s0, '
               posts', 'id')})
            outputs.extend(util.get_data(s2, 'tags', 'name'))
        else:
            pass
    s0 = s0_all
    s3 = util.do_sql(conn, "SELECT `pages`.* FROM `pages`  ORDER BY
        title", {})
    outputs.extend(util.get_data(s3, 'pages', 'title'))
    outputs.extend(util.get_data(s3, 'pages', 'slug'))
    s4 = util.do_sql(conn, "SELECT `posts`.* FROM `posts` WHERE (`
       posts`.`published_at` IS NOT NULL)", {})
    s5 = util.do_sql(conn, "SELECT `tags`.* FROM `tags` WHERE 1=0",
        {})
    return util.add_warnings(outputs)
```

D.6.3 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.6.4 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):
        s1 = util.do_sql(conn, "SELECT  `pages`.* FROM `pages`
           ORDER BY created_at DESC LIMIT 30 OFFSET 0", {})
        outputs.extend(util.get_data(s1, 'pages', 'id'))
        outputs.extend(util.get_data(s1, 'pages', 'title'))
        outputs.extend(util.get_data(s1, 'pages', 'slug'))
        outputs.extend(util.get_data(s1, 'pages', 'body'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.6.5 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.6.6 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):
        s1 = 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(s1, 'posts', 'id'))
        outputs.extend(util.get_data(s1, 'posts', 'title'))
        outputs.extend(util.get_data(s1, 'posts', 'body'))
        s1_all = s1
        for s1 in s1_all:
            s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `comments`
                 WHERE `comments`.`post_id` = :x0", {'x0': util.
               get_one_data(s1, 'posts', 'id')})
        s1 = s1_all
        pass
    else:
        pass
    return util.add_warnings(outputs)
```

D.6.7 Command get_admin

For this command we use a trimmed version of Enki, where we removed a `<ul>` element that displays recent comments in the file `app/views/admin/dashboard/show.html.erb`.

```
def get_admin (conn, inputs):
    util.clear_warnings()
```

```
    outputs = []
    s0 = util.do_sql(conn, "SELECT  posts.*, max(comments.
       created_at), comments.post_id FROM `posts` INNER JOIN
       comments ON comments.post_id = posts.id GROUP BY comments.
       post_id, posts.id, posts.title, posts.slug, posts.body,
       posts.body_html, posts.active, posts.approved_comments_count
       , posts.cached_tag_list, posts.published_at, posts.
       created_at, posts.updated_at, posts.edited_at  ORDER BY max(
       comments.created_at) desc LIMIT 5", {})
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT  `comments`.* FROM `comments
           ` WHERE `comments`.`post_id` = :x0  ORDER BY created_at
           DESC LIMIT 1", {'x0': util.get_one_data(s0, 'posts', 'id
           ')})
    s0 = s0_all
    s2 = util.do_sql(conn, "SELECT COUNT(*) FROM `posts`", {})
    s3 = util.do_sql(conn, "SELECT COUNT(*) FROM `comments`", {})
    s4 = util.do_sql(conn, "SELECT COUNT(*) FROM `tags`", {})
    s5 = util.do_sql(conn, "SELECT  `posts`.* FROM `posts` WHERE
       (1=1)  ORDER BY published_at DESC LIMIT 8", {})
    outputs.extend(util.get_data(s5, 'posts', 'id'))
    outputs.extend(util.get_data(s5, 'posts', 'title'))
    outputs.extend(util.get_data(s5, 'posts', 'slug'))
    s5_all = s5
    for s5 in s5_all:
        s6 = util.do_sql(conn, "SELECT COUNT(*) FROM `comments`
           WHERE `comments`.`post_id` = :x0", {'x0': util.
           get_one_data(s5, 'posts', 'id')})
    s5 = s5_all
    pass
    return util.add_warnings(outputs)
```

D.7 Regenerated Code for Blog

D.7.1 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'))
    return util.add_warnings(outputs)
```

D.7.2 Command get_article_id

```
def get_article_id (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT  `articles`.* FROM `articles`
       WHERE `articles`.`id` = :x0 LIMIT 1", {'x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'articles', 'id'))
    outputs.extend(util.get_data(s0, 'articles', 'title'))
    outputs.extend(util.get_data(s0, 'articles', 'text'))
    if util.has_rows(s0):
        s1 = util.do_sql(conn, "SELECT `comments`.* FROM `comments`
            WHERE `comments`.`article_id` = :x0", {'x0': inputs
           [0]})
        outputs.extend(util.get_data(s1, 'comments', 'commenter'))
        outputs.extend(util.get_data(s1, 'comments', 'body'))
        outputs.extend(util.get_data(s1, 'comments', 'article_id'))
    else:
        pass
    return util.add_warnings(outputs)
```

D.8 Regenerated Code for Student Registration System

D.8.1 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):
        s1 = util.do_sql(conn, "SELECT * FROM student WHERE id=:x0
           AND password=:x1", {'x0': inputs[0], 'x1': inputs[1]})
        if util.has_rows(s1):
            s2 = util.do_sql(conn, "SELECT * FROM course c JOIN
               registration r on r.course_id = c.id WHERE r.
               student_id = :x0", {'x0': inputs[0]})
            outputs.extend(util.get_data(s2, 'course', 'id'))
            outputs.extend(util.get_data(s2, 'course', 'teacher_id'
               ))
            outputs.extend(util.get_data(s2, 'registration', '
               course_id'))
            s2_all = s2
            for s2 in s2_all:
                s3 = util.do_sql(conn, "Select firstname, lastname
                   from teacher where id = :x0", {'x0': util.
                   get_one_data(s2, 'course', 'teacher_id')})
                s4 = util.do_sql(conn, "SELECT count(*) FROM
                   registration WHERE course_id = :x0", {'x0': util
                   .get_one_data(s2, 'course', 'id')})
            s2 = s2_all
            pass
        else:
            pass
    else:
```

```
        pass
    return util.add_warnings(outputs)
```

D.9 Regenerated Code for Synthetic

D.9.1 Command repeat_2

```
def repeat_2 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    return util.add_warnings(outputs)
```

D.9.2 Command repeat_3

```
def repeat_3 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s3 = util.do_sql(conn, "SELECT * FROM t2", {})
    return util.add_warnings(outputs)
```

D.9.3 Command repeat_4

```
def repeat_4 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
```

```
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s3 = util.do_sql(conn, "SELECT * FROM t2", {})
    s4 = util.do_sql(conn, "SELECT * FROM t2", {})
    return util.add_warnings(outputs)
```

D.9.4 Command repeat_5

```
def repeat_5 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s1 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s3 = util.do_sql(conn, "SELECT * FROM t2", {})
    s4 = util.do_sql(conn, "SELECT * FROM t2", {})
    s5 = util.do_sql(conn, "SELECT * FROM t2", {})
    return util.add_warnings(outputs)
```

D.9.5 Command nest

```
def nest_2 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t0", {})
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT * FROM t1", {})
        s1_all = s1
        for s1 in s1_all:
            s2 = util.do_sql(conn, "SELECT * FROM t2", {})
        s1 = s1_all
        pass
    s0 = s0_all
    pass
    return util.add_warnings(outputs)
```

D.9.6 Command after_2

```
def after_2 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s0 = s0_all
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2_all = s2
    for s2 in s2_all:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = s2_all
    pass
    return util.add_warnings(outputs)
```

D.9.7 Command after_3

```
def after_3 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s0 = s0_all
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2_all = s2
    for s2 in s2_all:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = s2_all
    s4 = util.do_sql(conn, "SELECT * FROM t3", {})
    s4_all = s4
    for s4 in s4_all:
```

```
        s5 = util.do_sql(conn, "SELECT * FROM t0", {})
    s4 = s4_all
    pass
    return util.add_warnings(outputs)
```

D.9.8 Command after_4

```
def after_4 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s0 = s0_all
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2_all = s2
    for s2 in s2_all:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = s2_all
    s4 = util.do_sql(conn, "SELECT * FROM t3", {})
    s4_all = s4
    for s4 in s4_all:
        s5 = util.do_sql(conn, "SELECT * FROM t0", {})
    s4 = s4_all
    s6 = util.do_sql(conn, "SELECT * FROM t4", {})
    s6_all = s6
    for s6 in s6_all:
        s7 = util.do_sql(conn, "SELECT * FROM t0", {})
    s6 = s6_all
    pass
    return util.add_warnings(outputs)
```

D.9.9 Command after_5

```
def after_5 (conn, inputs):
    util.clear_warnings()
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM t1", {})
    s0_all = s0
    for s0 in s0_all:
        s1 = util.do_sql(conn, "SELECT * FROM t0", {})
    s0 = s0_all
    s2 = util.do_sql(conn, "SELECT * FROM t2", {})
    s2_all = s2
    for s2 in s2_all:
        s3 = util.do_sql(conn, "SELECT * FROM t0", {})
    s2 = s2_all
    s4 = util.do_sql(conn, "SELECT * FROM t3", {})
    s4_all = s4
    for s4 in s4_all:
        s5 = util.do_sql(conn, "SELECT * FROM t0", {})
    s4 = s4_all
    s6 = util.do_sql(conn, "SELECT * FROM t4", {})
    s6_all = s6
    for s6 in s6_all:
        s7 = util.do_sql(conn, "SELECT * FROM t0", {})
    s6 = s6_all
    s8 = util.do_sql(conn, "SELECT * FROM t5", {})
    s8_all = s8
    for s8 in s8_all:
        s9 = util.do_sql(conn, "SELECT * FROM t0", {})
    s8 = s8_all
    pass
    return util.add_warnings(outputs)
```

D.9.10 Command `example` (Section 6.1)

```
def example_3 (conn, inputs):
    util.clear_warnings()
```

```
    outputs = []
    s0 = util.do_sql(conn, "SELECT * FROM tasks WHERE id = :x0", {'
       x0': inputs[0]})
    outputs.extend(util.get_data(s0, 'tasks', 'title'))
    if util.has_rows(s0):
        s1 = util.do_sql(conn, "SELECT * FROM comments WHERE
           task_id = :x0", {'x0': inputs[0]})
        outputs.extend(util.get_data(s1, 'comments', 'content'))
        s1_all = s1
        for s1 in s1_all:
            s2 = util.do_sql(conn, "SELECT * FROM users WHERE id =
               :x0", {'x0': util.get_one_data(s1, 'comments', '
               commenter_id')})
            outputs.extend(util.get_data(s2, 'users', 'name'))
            s3 = util.do_sql(conn, "SELECT * FROM tasks WHERE
               creator_id = :x0", {'x0': util.get_one_data(s1, '
               comments', 'commenter_id')})
            outputs.extend(util.get_data(s3, 'tasks', 'title'))
        s1 = s1_all
        s4 = util.do_sql(conn, "SELECT * FROM users WHERE id = :x0"
           , {'x0': util.get_one_data(s0, 'tasks', 'assignee_id')})
        outputs.extend(util.get_data(s4, 'users', 'name'))
        s5 = util.do_sql(conn, "SELECT * FROM tasks WHERE
           creator_id = :x0", {'x0': util.get_one_data(s0, 'tasks',
            'assignee_id')})
        outputs.extend(util.get_data(s5, 'tasks', 'title'))
    else:
        pass
    return util.add_warnings(outputs)
```

Bibliography

[1] Stack Overflow. http://stackoverflow.com/.

[2] Enki. `https://github.com/xaviershay/enki`, 2018. Git commit hash e70af252596f4123761bf389d371fe1b73611b73.

[3] Fulcrum. `https://github.com/fulcrum-agile/fulcrum`, 2018. Git commit hash dd4e59ec24c6f45d018fd2436c01836a04b15fef.

[4] Getting started with rails. `http://guides.rubyonrails.org/getting_started.html`, 2018.

[5] Kandan – modern open source chat. `https://github.com/kandanapp/kandan`, 2018. Git commit hash 380efafdf220fb7b14a097867a333b37d86e9c0b.

[6] Pldi 2019 konure code. `http://people.csail.mit.edu/jiasi/pldi2019.code/` and `https://people.csail.mit.edu/rinard/paper/pldi2019.code/`, 2019.

[7] Software assurance reference dataset. `https://samate.nist.gov/SARD/testsuite.php`, 2020.

[8] Railscollab: A project management and collaboration tool inspired by basecamp. `https://github.com/jamesu/railscollab/`, 2021. Git commit hash 9f6c8c12c990e39411e060879589006bde086ace.

[9] Todo: Basic rails gtd app. `https://github.com/engineyard/todo`, 2021. Git commit hash af9bc2f6e95452c23e9cb2cdb25f8427a41f998c.

[10] Benchmark application source code. `http://people.csail.mit.edu/jiasi/thesis2022.code/`, 2022.

[11] Kanban. `https://github.com/seanomlor/kanban`, 2022. Git commit hash 0660e77aa3734210d06aa2c8e0a444798e8eb862.

[12] F. Aarts, J. De Ruiter, and E. Poll. Formal models of bank cards for free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468, March 2013.

[13] Fides Aarts and Frits Vaandrager. *Learning I/O Automata*, pages 71–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1205–1220, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6), nov 2019.

[16] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[17] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Wafa: Fine-grained dynamic analysis of web applications. In *2009 11th IEEE International Symposium on Web Systems Evolution*, pages 141–150, Sept 2009.

[18] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 934–950, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] Manoli Albert, Jordi Cabot, Cristina Gómez, and Vicente Pelechano. Automatic generation of basic behavior schemas from uml class diagrams. *Software & Systems Modeling*, 9(1):47–67, 2010.

[20] 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. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[21] P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard. Program fracture and recombination for efficient automatic code reuse. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2015.

[22] Kijin An and Eli Tilevich. *Client Insourcing: Bringing Ops In-House for Seamless Re-Engineering of Full-Stack JavaScript Applications*, page 179–189. Association for Computing Machinery, New York, NY, USA, 2020.

[23] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[24] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269, September 1983.

[25] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Preventing sql injection attacks using dynamic candidate evaluations.

In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 12–24, New York, NY, USA, 2007. ACM.

[26] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to simd loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 123–134, New York, NY, USA, 2013. ACM.

[27] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 95–110, 2017.

[28] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 678–692, New York, NY, USA, 2018. ACM.

[29] Michael Bayer. Sqlalchemy. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.

[30] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[31] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. Recursive games for compositional program synthesis. In *Verified Software: Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, pages 19–39, 2015.

[32] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. Comput.*, 24(2):122–136, February 1975.

[33] Marina Billes, Anders Møller, and Michael Pradel. Systematic black-box analysis of collaborative web applications. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 171–184, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):14:1–14:39, March 2010.

[35] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. Interactive join query inference with jim. *Proc. VLDB Endow.*, 7(13):1541–1544, August 2014.

[36] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 195–205, New York, NY, USA, 2004. ACM.

[37] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.

[38] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[39] Cristian Buciluă, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. ACM.

[40] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, Nov 2005.

[41] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 621–634, New York, NY, USA, 2009. ACM.

[42] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Computer Networks*, 57(2):451 – 474, 2013. Botnet Activity: Analysis, Detection and Shutdown.

[43] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

[44] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[45] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. Active learning for software engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 62–78, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Softw. Syst. Model.*, 13(2):713–734, May 2014.

[47] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 609–633. Springer-Verlag, 2011.

[48] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016.

[49] Geetam Chawla, Navneet Aman, Raghavan Komondoor, Ashish Bokil, and Nilesh Kharat. Verification of orm-based controllers by summary inference. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2340–2351, 2022.

[50] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. Web question answering with neurosymbolic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 328–343, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 487–502, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[53] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978.

[54] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[55] Jürgen Cito, Jiasi Shen, and Martin Rinard. An empirical study on the impact of deimplicitization on comprehension in programs using application frameworks. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 598–601, 2020. Registered report.

[56] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. Dynamic program analysis for database reverse engineering. In *Generative and Transformational Techniques in Software Engineering IV*, pages 297–321. Springer, 2013.

[57] Yossi Cohen and Yishai A. Feldman. Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM Trans. Softw. Eng. Methodol.*, 12(3):285–316, July 2003.

[58] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng.*, 35(5):684–702, September 2009.

[59] Mark W. Craven and Jude W. Shavlik. Extracting tree-structured representations of trained networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, pages 24–30, Cambridge, MA, USA, 1995. MIT Press.

[60] Kathi Hogshead Davis and Peter H. Aiken. Data reverse engineering: A historical survey. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 70–, Washington, DC, USA, 2000. IEEE Computer Society.

[61] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 193–206, Berkeley, CA, USA, 2015. USENIX Association.

[63] Django. The web framework for perfectionists with deadlines | django. https://www.djangoproject.com/, 2018.

[64] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. Webrobot: Web robotic process automation using interactive programming-by-demonstration. In *Proceedings of the 43th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2022, 2022.

[65] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1289–1297, 2016.

[66] E. N. Elnozahy. Address trace compression through loop detection and reduction. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, pages 214–215, New York, NY, USA, 1999. ACM.

[67] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435, 2018.

[68] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 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 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[69] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[70] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. *Combining Model Learning and Model Checking to Analyze TCP Implementations*, pages 454–471. Springer International Publishing, Cham, 2016.

[71] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1 of *COMPSAC 2007*, pages 87–96, July 2007.

[72] Rubén Fuentes-Fernández, Juan Pavón, and Francisco Garijo. A model-driven process for the modernization of component-based systems. *Sci. Comput. Program.*, 77(3):247–269, March 2012.

[73] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[74] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. Learning commutativity specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 307–323, 2015.

[75] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[76] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[77] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, New York, NY, USA, 2012. ACM.

[78] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theor. Comput. Sci.*, 411(47):4029–4054, October 2010.

[79] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, page 13–24, New York, NY, USA, 2010. Association for Computing Machinery.

[80] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[81] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM.

[82] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[83] B. P. Gupta, D. Vira, and S. Sudarshan. X-data: Generating test data for killing sql mutants. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 876–879, March 2010.

[84] Raju Halder and Agostino Cortesi. Obfuscation-based analysis of sql injection attacks. In *Proceedings of the The IEEE Symposium on Computers and Communications*, ISCC '10, pages 931–938, Washington, DC, USA, 2010. IEEE Computer Society.

[85] William G. J. Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 174–183, New York, NY, USA, 2005. ACM.

[86] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. Improving the performance of trace-based systems by false loop filtering. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 405–418, New York, NY, USA, 2011. ACM.

[87] Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 710–720, 2015.

[88] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[89] Malte Isberner, Falk Howar, and Bernhard Steffen. *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning*, pages 307–322. Springer International Publishing, Cham, 2014.

[90] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 156–167, 2016.

[91] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Jsketch: sketching for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 934–937, 2015.

[92] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[93] Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. Guiding dynamic programing via structural probability for accelerating programming by example. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[94] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. ACM.

[95] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[96] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 711–726, New York, NY, USA, 2016. Association for Computing Machinery.

[97] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 94–103, New York, NY, USA, 2008. ACM.

[98] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012.

[99] M. Kobayashi. Dynamic characteristics of loops. *IEEE Trans. Comput.*, 33(2):125–132, February 1984.

[100] Paul Krogmeier, Umang Mathur, Adithya Murali, P. Madhusudan, and Mahesh Viswanathan. Decidable synthesis of programs with uninterpreted functions. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 634–657, Cham, 2020. Springer International Publishing.

[101] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[102] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, page 70–80, New York, NY, USA, 2016. Association for Computing Machinery.

[103] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *SIGPLAN Not.*, 53(4):436–449, June 2018.

[104] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[105] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press.

[106] Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. Active refinement of clone anomaly reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 397–407. IEEE Press, 2012.

[107] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA, 2015. ACM.

[108] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[109] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages 143–152, New York, NY, USA, 2007. ACM.

[110] Nesrine Noughi, Marco Mori, Loup Meurice, and Anthony Cleve. Understanding the database manipulation behavior of programs. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 64–67, New York, NY, USA, 2014. ACM.

[111] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[112] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. Synthesizing data structure refinements from integrity constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 574–587, New York, NY, USA, 2021. Association for Computing Machinery.

[113] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 408–418, New York, NY, USA, 2014. ACM.

[114] Jeff Perkins, Jordan Eikenberry, Alessandro Coglio, Daniel Willenson, Stelios Sidiroglou-Douskos, and Martin Rinard. Autorand: Automatic keyword randomization to prevent injection attacks. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 37–57, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[115] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. Assertion generation through active learning. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 155–157, Piscataway, NJ, USA, 2017. IEEE Press.

[116] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538, 2016.

[117] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Kaelbling. Selecting representative examples for program synthesis. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4161–4170. PMLR, 10–15 Jul 2018.

[118] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury. Modeling software execution environment. In *2012 19th Working Conference on Reverse Engineering*, pages 415–424, Oct 2012.

[119] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černý. Droidstar: Callback typestates for android classes. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1160–1170, New York, NY, USA, 2018. ACM.

[120] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '05, pages 62–71, New York, NY, USA, 2005. ACM.

[121] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[122] Ruby On Rails. Ruby on rails | a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. https://rubyonrails.org/, 2018.

[123] George Reese. *Database Programming with JDBC and JAVA*. O'Reilly Media, Inc., 2000.

[124] Alex Renda, Yi Ding, and Michael Carbin. Programming with neural surrogates of programs. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2021, page 18–38, New York, NY, USA, 2021. Association for Computing Machinery.

[125] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, pages 611–622, 2007.

[126] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. Active learning for inference and regeneration of computer programs that store and retrieve data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 12–28, New York, NY, USA, 2018. ACM.

[127] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 139–149, New York, NY, USA, 2016. ACM.

[128] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engg.*, 21(2):147–186, April 2014.

[129] Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 25:1–25:10, New York, NY, USA, 2011. ACM.

[130] Stephen R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2007.

[131] Douglas C Schmidt. Model-driven engineering. 2006.

[132] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[133] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[134] Jiasi Shen and Martin Rinard. Using dynamic monitoring to synthesize models of applications that access databases. Technical report, September 2018. http://hdl.handle.net/1721.1/118184.

[135] Jiasi Shen and Martin Rinard. 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, 2019.

[136] Jiasi Shen and Martin Rinard. Active loop detection for applications that access databases. Technical report, November 2021. https://hdl.handle.net/1721.1/138144.

[137] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 431–432, New York, NY, USA, 2022. Association for Computing Machinery.

[138] Jiasi Shen and Martin C. Rinard. Active learning for inference and regeneration of applications that access databases. *ACM Trans. Program. Lang. Syst.*, 42(4), January 2021.

[139] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 515–527, New York, NY, USA, 2018. Association for Computing Machinery.

[140] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 473–486, New York, NY, USA, 2015. ACM.

[141] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 289–299, New York, NY, USA, 2011. ACM.

[142] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 326–340, New York, NY, USA, 2016. Association for Computing Machinery.

[143] Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In Francesco Ranzato, editor, *Static Analysis*, pages 364–381, Cham, 2017. Springer International Publishing.

[144] Sunbeom So and Hakjoo Oh. Synthesizing pattern programs from examples. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 1618–1624. AAAI Press, 2018.

[145] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[146] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 313–326, New York, NY, USA, 2010. Association for Computing Machinery.

[147] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5):497–518, 2013.

[148] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. Synthesizing switching logic using constraint solving. *International Journal on Software Tools for Technology Transfer*, 13(6):519–535, 2011.

[149] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. Tesma and catg: Automated test generation tools for models of enterprise applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 717–720, May 2015.

[150] Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Mach. Learn.*, 13(1):71–101, October 1993.

[151] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618, Austin, TX, 2016. USENIX Association.

[152] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pages 14–23, Feb 1998.

[153] Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, January 2017.

[154] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin Rinard. Supply-chain vulnerability elimination via active learning and regeneration. 2021. ACM Conference on Computer and Communications Security.

[155] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering*, pages 49–68, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[156] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic sql query explorer. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 425–446, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[157] Michele Volpato and Jan Tretmans. Approximate active learning of nondeterministic input output transition systems. *Electronic Communications of the EASST*, 72, 2015.

[158] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466, New York, NY, USA, 2017. ACM.

[159] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 452–466, New York, NY, USA, 2017. Association for Computing Machinery.

[160] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Speeding up symbolic reasoning for relational queries. *Proc. ACM Program. Lang.*, 2(OOPSLA):157:1–157:25, October 2018.

[161] Shaowei Wang, David Lo, and Lingxiao Jiang. 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, page 677–682, New York, NY, USA, 2014. Association for Computing Machinery.

[162] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[163] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[164] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.

[165] Michael Widenius and Davis Axmark. *Mysql Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[166] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971.

[167] Jerry Wu. Using dynamic analysis to infer python programs and convert them into database programs. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2018.

[168] Jerry Wu. Using dynamic analysis to infer python programs and convert them into database programs. Master's thesis, Massachusetts Institute of Technology, 2018. `https://hdl.handle.net/1721.1/121643`.

[169] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 29–35, New York, NY, USA, 2016. ACM.

[170] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 508–521, 2016.

[171] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593, January 2018.

[172] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, October 2017.

[173] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 1299–1308, New York, NY, USA, 2017. ACM.

[174] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 470–479, Washington, DC, USA, 2004. IEEE Computer Society.

[175] Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig, and Greg Durrett. Sketch-Driven Regular Expression Generation from Natural Language and Examples. *Transactions of the Association for Computational Linguistics*, 8:679–694, 11 2020.