Using Dynamic Analysis to Infer Python Programs and Convert Them into Database Programs

by

Jerry Wu

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author .................................................................
Department of Electrical Engineering and Computer Science
June 7, 2018

Certified by ...........................................................
Martin Rinard
Professor
Thesis Supervisor

Accepted by ...........................................................
Katrina LaCurts
Chair, Master of Engineering Thesis Committee
Using Dynamic Analysis to Infer Python Programs and Convert Them into Database Programs

by

Jerry Wu

Submitted to the Department of Electrical Engineering and Computer Science on June 7, 2018, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

I present NERO, a new system that automatically infers and regenerates programs that access databases. The developer first implements a Python program that uses lists and dictionaries to implement the database functionality. NERO then instruments the Python list and dictionary implementations and uses active learning to generate inputs that enable it to infer the behavior of the program. The program can be implemented in any arbitrary style as long as it implements behavior expressible in the domain specific language that characterizes the behaviors that NERO is designed to infer. The regenerated program replaces the Python lists and dictionaries with database tables and contains all code required to successfully access the databases. Results from several inferred and regenerated applications highlight the ability of NERO to enable developers with no knowledge of database programming to obtain programs that successfully access databases.

Thesis Supervisor: Martin Rinard
Title: Professor
Acknowledgments

First and foremost, I would like to thank my advisor, Professor Martin Rinard, for giving me the opportunity to do my thesis under him when I least expected it, and for his guidance, support, and encouragement throughout my research process. I would also like to thank Jiasi Shen for giving me extensive guidance and feedback both when I performed research as an undergraduate and when helping me revise this thesis. I would like to thank Anne Hunter for her tremendous patience in helping me with all the procedures I needed to go through in order to complete my thesis in a timely manner. I would like to thank Aofei Liu for her kindness and emotional support throughout my research and writing process, in both good times and bad. Finally, I could not have performed my work or written my thesis without the unwavering support of friends and family. To them I am eternally grateful.
Contents

List of Figures 11

1 Introduction 13
  1.1 Use Cases 14
  1.2 Supported Seed Programs 15
  1.3 Inference Algorithm Overview 17
  1.4 Regeneration Algorithm Overview 19
  1.5 Contributions 20

2 Example 23
  2.1 NERO Input Components and Output Example 23
    2.1.1 Seed Program 24
    2.1.2 Command Specification Text File 25
    2.1.3 Data Specification Text File 28
    2.1.4 Sample Data and Execution 32
    2.1.5 Corresponding Regenerated Program 35
  2.2 Supported Seed Programs 38
  2.3 Inference 40
    2.3.1 Initialization and Preprocessing 40
    2.3.2 Inferring Data Sources for register: 42
    2.3.3 Inferring Loops for list_student_courses: 48
    2.3.4 Ambiguity in the Inferred Representation 53
  2.4 Regeneration 55
3 Nero Domain Specific Language

3.1 Grammar of the Nero DSL

3.1.1 Variable Categories
3.1.2 Nonterminal Symbols
3.1.3 Variable Restrictions

3.2 Scope

3.3 Syntactic Flexibility

4 Seed Program Interface

4.1 Command Interface Generation
4.2 Data Structure Manipulation and Translation

5 Data Structure Instrumentation

5.1 Recording Function Calls
5.2 Recording Loops
5.3 Recording Crashes
5.4 Implementation

6 Inference Algorithm

6.1 Learning UIDs of Data Structures
6.2 Choosing Values for Inputs and Data Structures
6.3 Constraints to Avoid Crashes
6.4 Identifying Data Sources

6.4.1 Resolving ambiguity
6.5 Identifying Loops
6.6 Discovering Conditional Blocks in Loops

6.6.1 Allowing Unknown Conditions to Satisfy
6.6.2 Using Temperature to Reduce Ambiguity
6.7 Inferred Representation
6.8 Correctness .................................................. 105
6.8.1 Execution of all code paths .............................. 106
6.8.2 Correct inference of sequence on execution .......... 107
6.8.3 Determining a superset of input argument sources ... 108
6.8.4 Determining a superset of conditions ................. 109
6.8.5 Eliminating false ambiguity with high probability ... 109
6.8.6 Correctness Analysis Summary ......................... 134

7 Regeneration for Database Backend ........................ 137
7.1 Regeneration Algorithm ................................... 138
7.1.1 Regenerating the Database Schema .................... 138
7.1.2 Regenerating Inferred Commands ..................... 139
7.1.3 Regenerating Boilerplate .............................. 143
7.2 Regenerating Program Behavior ......................... 144

8 Experimental Results ......................................... 147
8.1 Task management application ............................. 148
8.1.1 get_api_board ........................................... 151
8.1.2 get_api_cards_id ....................................... 154
8.1.3 post_api_lists and post_api_cards .................... 154
8.2 Chat room application ..................................... 155
8.3 Blog and student registration applications ............... 157
8.4 Evaluation .................................................. 158
8.5 Discussion .................................................. 158

9 Future work ................................................... 161
9.1 Interface Improvements ................................... 162
9.2 Dict String Keys ........................................... 162
9.3 Dict items() ................................................. 164
9.4 Value Return ............................................... 165
9.5 Regenerated Code Optimizations ........................ 167
9.6 Regenerated Code Readability ........................................ 168
9.7 Different Regeneration Target ................................. 169
9.8 Supporting More Query Operations ....................... 170
9.9 Analytical Temperature Selection Algorithm ...................... 172
9.10 Analytical Input Solver ........................................... 173

10 Related Work .......................................................... 175

11 Conclusion ............................................................... 177
  11.1 Nero Key Aspects .................................................. 177
  11.2 Implications .......................................................... 178

A Code ........................................................................ 181

Bibliography .................................................................. 195
List of Figures

2-1 Seed program for student registration application . . . . . . . . . . . 26
2-2 Command specification file for student registration application . . 27
2-3 Data specification file for the student registration application . . . 31
2-4 Sample data for the student registration application example . . . 32
2-5 Sample sequence of executed commands for the student registration
application example . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2-6 Sample execution result for the student registration application example 34
2-7 Regenerated database schema for student registration application . 36
2-8 Regenerated list_student_courses command for the student registr-
ation application. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
2-9 IR of the register command inferred by NERO. . . . . . . . . . . 48
2-10 IR of the list_student_courses command inferred by NERO. . . . 53

3-1 Abstract syntax for seed programs . . . . . . . . . . . . . . . . . . . . 62

6-1 Inference algorithm outline . . . . . . . . . . . . . . . . . . . . . . 86

8-1 Data specification file for the task management application . . . . 149
8-2 Regenerated database schema for task management application . . 150
8-3 Command specification file for task management application . . . 150
8-4 Data specification file for the chat room application . . . . . . . 156
8-5 Regenerated database schema for chat room application . . . . . . 156
8-6 Command specification file for the chat room application . . . . . 156

A-1 Seed program for the task management application. . . . . . . . . 182
A-2 Regenerated program for the task management application. . . . . . . 183
A-3 Seed program for task management application (continued). . . . . . 184
A-4 Regenerated program for the task management application (continued). 185
A-5 Seed program for the task management application (continued). . . . 186
A-6 Regenerated program for the task management application (continued). 187
A-7 Seed program for the chat room application. . . . . . . . . . . . . . 188
A-8 Regenerated program for the chat room application. . . . . . . . . . 189
A-9 Regenerated program for the chat room application (continued). . . 190
A-10 Seed program for the blog application. . . . . . . . . . . . . . . . 191
A-11 Regenerated program for the blog application. . . . . . . . . . . . 192
A-12 Regenerated program for the student registration application. . . . 193
Chapter 1

Introduction

Programs which access databases are used in a wide array of applications, but developers, especially novice developers, who face the task of writing them may not necessarily be proficient with SQL. I present a new approach which allows these developers to implement the functionality of the database program in Python, an easy-to-learn, easy-to-write, and widely taught language. The developer writes a seed program in Python 3, using standard Python lists and dictionaries stored in global variables to emulate the functionality of the database. My system, Nero, instruments the Python list and dict implementations, then runs the seed program in the instrumented environment. Nero uses an active learning approach, where it dynamically generates inputs to the seed program and populates the contents of the data structures, and observes the resulting sequence of global data structure accesses, to infer the functionality implemented by the seed program, without analyzing the source code of the seed program. After Nero infers the seed program’s functionality, it then automatically regenerates an optimized Python program implementing the same functionality, but using a SQLite database in place of lists and dicts. The regenerated program also contains automatically generated input validation code designed to check for corner cases that would otherwise cause the program to crash.
1.1 Use Cases

NERO allows the user to write a seed program performing a sequence of operations to look up or store information using Python data structures. NERO subsequently infers the functionality of the seed program and produces a regenerated program implementing the same sequence of operations using a database instead of Python data structures. NERO thereby allows a developer to focus on implementing the desired functionality without needing to think about SQL syntax, query efficiency, or input validation. I anticipate several use cases:

- **Novice Software Developers**: In this use case, someone who is new to software development and has recently learned how to write Python may wish to develop a database-backed application, e.g. for a personal project. NERO will allow them to develop their application without needing to learn SQL.

- **Specialists in Other Fields**: In this use case, a specialist in a field other than computer science (such as a biologist or chemist) may develop to write a small database-backed application to process or store experimental data. If the specialist knows how to code in Python, which is easy to learn and widely taught in schools, they can use NERO to develop their application without needing to learn SQL. NERO will allow scientists to quickly develop custom software to store their data without needing to either store their data in a spreadsheet (which does not scale well to large amounts of data) or find or purchase existing software.

- **Rapid Prototyping**: In this use case, a software company may wish to develop a prototype for a new application. Although developers at the company may already be acquainted with SQL, writing a database-backed application may require substantial amounts of time spent searching for the correct syntax on Google. In contrast, NERO seed programs can be written very quickly in Python, allowing the company to rapidly build and test new prototypes. For example, I wrote each of the four seed programs for the benchmark applications discussed in
Chapter 8 in approximately half an hour or less. These benchmark applications implement core functionality representative of the types of programs users may wish to develop using Nero.

In the longer term, I anticipate Nero’s functionality will extend in multiple directions, leading to a broader set of use cases.

- I expect Nero’s inference algorithm to be expanded to support a wider range of seed programs. Expanding Nero’s inference algorithm will give users more flexibility in the functionality of database-backed programs they wish to develop.

- I expect Nero’s regeneration algorithm to be expanded to more complex regeneration targets. For example, Nero can be modified to regenerate a program in Ruby (or any other language) backed by a distributed database such as Druid [2] (or any other type of database). Regenerating a program in Ruby backed by a distributed database will allow a developer without a strong grasp of Ruby or distributed databases to easily develop a fast and efficient program. A developer could thereby use Nero to replace a critical module of an existing program with a more efficient implementation.

Next, I discuss in more detail what seed programs are currently supported by Nero.

1.2 Supported Seed Programs

To render the inference program tractable, Nero is designed to infer programs whose behavior can be modeled by a program in a domain-specific language which captures common usage patterns. Seed programs written in this manner use a set of global data structures to mimic databases and implement a set of commands, which perform operations on these global data structures.

Global data structures used by supported seed programs are lists or dicts which emulate the functionality of a database by storing tuples. Each tuple within a single
data structure must have the same length and type signature. In this manner, each
tuple represents a row of a database, and each position in a tuple represents a column
of a database. A list of tuples represents a database with an integer primary key (i.e.,
an id for each row) and a dict of tuples represents a database with a string primary
key.

The implementation of each command consists of a sequence of accesses to these
data structures which correspond to database operations. For example, a command
might perform a dict key lookup operation, which corresponds to a SELECT query
which contains a WHERE clause conditioning on the primary key. Or, a command might
perform a list index assignment operation, which corresponds to a UPDATE query with
a WHERE clause conditioning on the integer primary key. Commands can also use
print statements interspersed between data structure accessed to output values.

Commands are also allowed to perform sequences of data structure accesses while
iterating over data structures. On a basic level, this corresponds to performing a
SELECT query to retrieve the contents of a database table, then iterating over the
result and performing operations on it (such as printing each row of the result, or
performing queries based on each row of the result). Iterations are also allowed to
execute conditionally in a conditional loop (i.e., the main body of the loop is allowed to
be an if-statement). Making a loop conditional corresponds to adding a WHERE clause
to the SELECT query, thereby retrieving only some of the rows of the database table
instead of every row. Loops can be used in a nested manner; for example, if a loop
contains a nested conditional loop, this (depending on the condition) may correspond
to performing a JOIN operation.

The domain-specific language (DSL) Nero uses to represent seed program behavior
ensures tractability of the inference problem by constraining the seed program’s loops
and conditionals to the use cases described above, allowing Nero to force the seed
program to execute all code paths and thereby infer the seed program’s full functionality.
At the same time, Nero’s DSL is sufficiently expressive to allow the user to emulate
common database access patterns within the seed program, as demonstrated by the
experimental results.
Note that because NERO interacts with the seed program only via the inputs, outputs, and instrumented list and dict data structures, there is no need for NERO to analyze the source code of the seed program. The seed program can therefore be implemented in any style or coding pattern as long as its behavior lies within the scope of the DSL.

1.3 Inference Algorithm Overview

The inference algorithm infers the core functionality implemented by the seed program by dynamically generating inputs to the seed program and observing the resulting sequence of data structure accesses observed by NERO’s instrumentation.

Because the inference algorithm is designed to work with programs which implement only the core functionality desired by the user, the inference algorithm is designed to be able to infer the behavior of seed programs which do not perform any input validation to check for corner cases which may crash the seed program. For example, seed programs may crash due to an IndexError or KeyError when performing a list index or dict key lookup. When such an error occurs in the seed program, NERO automatically detects the error and restarts the seed program, and will generate data and inputs to the seed program which avoid the error in future executions. It is well-known that writing input validation code by hand is time-intensive and prone to error. Because the inference algorithm is able to infer the functionality implemented by seed programs which do not contain input validation code, the user is able to focus on implementing the core functionality they desire and is relieved of the burden of writing input validation and error-checking code.

NERO conceptualizes a seed program’s functionality as a sequence of data structure accesses. When NERO executes a seed program in the instrumented environment, the instrumentation captures the sequence of which data structures are accessed, the manner in which they are accessed (i.e., which accessor method was called), the concrete values of the arguments with which they are accessed, and the concrete values which are returned, if any. Because the sequence of accesses is directly given by the
instrumentation, the inference algorithm faces two remaining primary challenges to fully infer the seed program’s functionality:

1. To unambiguously determine the sources of data structure access arguments (e.g. “the first argument was a command parameter and the second argument was returned by the first data structure access in the command”)

2. Where conditionals exist, to infer the condition, and to ensure all code paths are executed so that the full functionality of the seed program can be inferred unambiguously

To tackle the first challenge, Nero ensures that when the seed program executes, the concrete values available to the seed program (i.e., command parameters and values stored in data structures) are as diverse as possible, so that directly matching concrete values observed by the instrumentation will allow Nero to unambiguously infer the sources of data structure access arguments.

To make the second challenge tractable, the domain-specific language (DSL) of seed programs supported by Nero is carefully structured so that Nero can readily infer whether each condition was satisfied or not in each execution. Although Nero’s instrumentation does not directly capture the presence of conditionals, because conditionals are only permitted at the entrance of a loop body, if the conditional does not execute, Nero will observe an empty iteration. Restricting conditionals to the entrance of a loop body is justified because in Nero’s DSL, conditionals are allowed only where they are necessary to express common data structure operations, such as SELECT queries with WHERE or JOIN clauses. Additionally, the DSL heavily constrains which conditions are allowed: they are only permitted to test for the equality of different values.

Restricting conditionals to only occur at a loop body entrance and check for equality allows Nero to intelligently populate data structures and choose command parameters to cause conditions to be satisfied. Because conditions only check for equality, Nero can cause conditions to satisfy by setting values in data structures or command parameters to be equal to each other. Because conditions are only present
at the entrance of a loop body, and the loop body is not allowed to be empty, NERO can detect the presence of an unsatisfied condition because the loop body observed in the concrete execution will be empty. Once conditions are satisfied at least once, NERO enumerates the set of possible conditions which are satisfied. The set of possible conditions is limited, and can be exhaustively enumerated, because only equality checks are allowed. Then, NERO repeatedly executes the seed program with different inputs and data structure contents, gradually pruning down the list of possible conditions, until the true condition has been inferred unambiguously. Note that NERO must balance between disambiguation and forcing of conditional satisfaction. Ambiguity arises when NERO provides the seed program with values which equal each other, but providing these equal values are necessary to cause equality-testing conditions to satisfy.

Having solved both of these challenges, NERO is able to infer a representation of the seed program’s functionality in an internal intermediate representation (IR), which specifies the sequence of data structure accesses and how the input arguments to each access are determined, including any loops or conditionals.

1.4 Regeneration Algorithm Overview

Once the inference algorithm has generated an IR representing the seed program’s functionality, NERO processes the IR and outputs a regenerated program written in Python implementing the same functionality as the seed program, but using SQLite databases in place of lists and dicts. Note that there is no fundamental reason the regenerated program must be a Python SQLite program; one could write a regeneration module for essentially any language which has a library for accessing databases, and attach the regeneration module to the same inference algorithm. The prototype NERO implementation regenerates to Python SQLite as a proof of concept, because Python SQLite is a simple way to write a database program.

Because NERO’s domain-specific language is designed specifically so that data structure accesses emulate database operations, it is a simple matter to read the IR
which stores the sequence of data structure accesses and convert each data structure access to the corresponding database query operation. NERO first determines the database schema needed for the regenerated program, by constructing the schema to match the data structures used by the seed program. Constructing the database schema to match the data structures in the seed program is a straightforward process because the data structures are written to mimic the structure of databases. NERO then converts the inference algorithm’s IR to a second IR, the SQL IR, which represents the functionality of the seed program as a sequence of database queries.

NERO performs optimizations on the SQL IR to improve the efficiency of the regenerated program. For example, as appropriate, NERO tries to combine nested loops that iterate over related tables using a JOIN operation. These optimizations are designed to increase the efficiency of the regenerated program.

After performing optimizations on the SQL IR, NERO compiles the SQL IR to a Python function which accesses a SQLite database to store and retrieve data. The regenerated Python function includes automatically generated input validation, which gracefully aborts the command and prints an error message if the command tried to retrieve a row of a database table which was not found. Finally, NERO generates the boilerplate code needed to generate the database tables and to parse user input from the command line.

In this manner, NERO is able to automatically infer a seed program’s functionality and regenerate an entire executable optimized Python program containing automatically generated input validation code and which uses SQLite databases instead of Python native data structures. Experimental results from my implementation demonstrate NERO’s ability to infer and regenerate programs which match the functionality of a variety of programs, including a student registration database application, a chat room application, a blog application, and a task management application.

1.5 Contributions

This thesis makes the following contributions:
• **Concept**: It introduces the concept of developing database programs by developing programs that use standard data structures such as lists and dictionaries, then inferring the behavior of the program to regenerate a program that replaces the lists and dictionaries with corresponding database tables.

• **Inference Algorithm**: It presents an algorithm that infers the behavior of the program by instrumenting the Python list and dictionary implementations, then running the program on test inputs and observing the resulting outputs and data structure interactions.

• **Results**: It presents experimental results that highlight the ability of Nero to infer and regenerate its target class of programs.

Note that some of the research presented in this thesis was performed in collaboration with Jiasi Shen and Professor Martin Rinard. Some of the contents of this thesis were also submitted to ASE 2018.
Chapter 2

Example

I present an example of this new approach of dynamic program analysis.

NERO infers the functionality of a seed program provided by the user, and outputs a regenerated program implementing the same functionality but using databases in place of data structures. In this example, I walk through the inference and regeneration of a student registration application. This program stores information on students (such as their names), information on courses (such as the course’s names), and registration information (who is registered for which courses). This program includes two commands which access the stored information: one to register students for new courses and one to retrieve information on a student and the courses they are registered for.

Below, I discuss the inputs the user provides to NERO, followed by the types of seed programs supported by NERO. Then, I walk through how NERO infers the functionality of the example seed program, followed by how NERO produces the regenerated program.

2.1 NERO Input Components and Output Example

To use NERO, the user provides a seed program implementing the desired functionality. However, because NERO does not analyze the seed program source code, NERO asks the user to provide two easy-to-write files indicating additional information:
• A command specification file providing information on the commands to be inferred.

• A data specification file providing information on the data structures used by the seed program and the desired database schema in the regenerated program.

I elaborate on the seed program and the two specification files below. I provide a set of sample data and a set of sample commands, and show the behavior of the seed program on the sample data with the sample commands. I also describe the regenerated program produced by NERO when given the example seed program, command specification file, and data specification file.

2.1.1 Seed Program

The user provides a seed program written in Python 3 containing a set of commands. Each command is implemented individually as a Python function which takes input arguments, accesses a shared set of global lists and dicts, and prints output to stdout. The seed program should additionally have the following properties:

• Each command should be implemented using only the subset of Python supported by the inference algorithm, or in a manner equivalent (in terms of both output and sequence of data structure accesses) to some such program. (Chapter 3)

• Initializes global data structures in a deterministic order, e.g., by declaring all of them at the top of the file.

Figure 2-1 shows the seed program for the student registration application. It defines three data structures at the top of the file: a list storing student information, a dict storing course information, and a list storing registration information. The seed program implements two commands. The first command, implemented by do_register, takes as input a student ID and a course number, prints the student’s name, and then adds a registration entry for the student to enroll in the course. The second command, implemented by do_liststudentcourses, takes as input a student
ID, prints the student’s name, retrieves registration information for the student, and
prints course information for each course this student is registered in.

As an additional note, it is straightforward to add additional functionality to
this seed program, for example implementing additional commands which add new
students or courses to the database, but these commands do not add explanatory
value to this example and are therefore not included.

2.1.2 Command Specification Text File

NERO uses a command specification text file to determine what Python functions
implement commands NERO should run inference on. NERO also uses this specification
file to generate a command loop which reads commands from stdin and dispatches
them to the corresponding Python functions in the seed program. NERO attaches the
command loop to the seed program so the commands can be executed by the inference
algorithm. The command loop is attached to the seed program in a preprocessing
step discussed in Section 2.3.1.

For each command implemented by the seed program, the specification file should
indicate:

- The name of the command. NERO will, in both the seed program and regenerated
  program, generate a command loop which reads commands from stdin and
  dispatches them to the corresponding Python functions. The name of the
  command is what the user will type to indicate which command is run.

- The name of the Python function in the seed program implementing the command.
The name of the Python function can be, but is not required to be, the same
as the name of the command. Specifying the name of the Python function
allows NERO to identify which function in the seed program implements which
command (if any). Including the name of the Python function for each command
in the command specification file allows NERO to invoke the correct function for
each command without analyzing the seed program source code.

25
# list of student information.
# index = student id, entries = (first_name, last_name)
student = []

# dict of course information.
# key = course number, entries = (name, teacher_id)
course = {}

# list of student registration information
# entries = (student_id, course number), indicating the student
# is registered for that course.
registration = []

def do_register(student_id_query, course_num_query):
    first_name, last_name = student[student_id_query]
    print(first_name, last_name)
    registration.append((student_id_query, course_num_query))

def do_liststudentcourses(student_id_query):
    first_name, last_name = student[student_id_query]
    print("Student: {}, {}".format(first_name, last_name))
    for reg_student_id, reg_course_num in registration:
        if student_id_query != reg_student_id:
            continue
        for course_num in course:
            if course_num == reg_course_num:
                course_name, course_teacher_id = course[course_num]
                print(course_num, course_name, course_teacher_id)

Figure 2-1: Seed program for the student registration application. Data structures are defined at the top of the file. Comments are provided for readability to specify the contents of each data structure, but are not necessary for Nero to run. The first command, implemented by do_register, first prints the specified student and then adds the student to the specified course. The second command, implemented by do_liststudentcourses, prints the specified student and then lists all courses the student is registered for. Note that the inner loop could be replaced by a simple dict lookup (course_name, course_teacher_id = course[reg_course_num]) followed by a print statement, but this example is a little more illustrative as to Nero's capabilities. Observe also that the commands are entirely independent of each other, except insofar as they access the same shared data structures.
Figure 2-2: Command specification file for the student registration application. Each line in the file describes a separate command. The first command, named register, is implemented by the do_register Python function, and takes two arguments: an integer and a string. The second command, named list_student_courses, is implemented by the do_liststudentcourses Python function, and takes a single integer argument. The order of commands in this file is not required to be the same as the order of the Python functions in the seed program, but for clarity they have been defined in this way.

- The signature of the command, i.e. the list of types of the command parameters. Command parameters are allowed to be integers (“int”) or strings (“str”). Specifying the signature of each command is necessary because NERO does not analyze the seed program source code to determine the expected types of input parameters.¹ NERO uses this information to determine what type of arguments to generate when executing the seed program during the inference algorithm.

For ease of parsing, the prototype implementation of NERO expects the command specification file to contain a JSON list, where each list entry corresponds to a single command. Each command is described by a list, containing first the name of the command (as a string), then the name of the Python function (as a string), then the types of each command parameter (as a string, either “str” or “int”). It would be straightforward to modify the NERO implementation to accept a more human-readable or human-writable command specification file instead.

Figure 2-2 presents the command specification file for the student registration application. The first command, named register, is implemented by the do_register Python function, and takes two arguments: an integer and a string. The second command, named list_student_courses, is implemented by the Python function do_liststudentcourses, and takes a single integer argument.

¹ Even if it did analyze the source code, Python is dynamically typed so identifying the argument types would be difficult.
2.1.3 Data Specification Text File

NERO uses a *data specification* text file to determine the structure of data it should populate each data structure with, as well as to determine the database schema in the regenerated program. Additionally, NERO uses this file to generate a set of *fill command* handlers, which are included in the command loop generated by NERO. Fill commands allow NERO to populate data structures with arbitrary contents during the inference process. The fill command handlers are generated and attached to the seed program during a preprocessing step discussed in Section 2.3.1.

The data specification file specifies the data structures used by the seed program and the form of these data structures, as well as the database schema in the regenerated program. Each data structure in the seed program mimics a database table; NERO will regenerate a database table for each data structure in the seed program. For each data structure and corresponding database table, this file specifies:

- The name of the data structure (more specifically, the name of the global variable storing the data structure in the seed program).
- Whether the data structure is a list or a dict
- The name of the corresponding database table in the regenerated program. The name of the table is allowed to be, but is not required to be, the same as the name of the data structure.
- The name and type of each column in the corresponding database table.

The data structures mimic database tables, as follows: Each data structure is either a list of tuples or a dict where the values are tuples. A tuple corresponds to a row in the corresponding database table, so the tuples within the same data structure are required to have the same length and type signature. Each position in the tuple corresponds to a column of the database table, and has the same type as its corresponding column. The database table also contains an additional column: a primary key. If the data structure is a list, the primary key column is of integer type.
and corresponds to the list index; if the data structure is instead a dict, the primary key column is of string type and corresponds to the dict key, which is of string type.\(^2\) (It follows that the number of columns in each database table is one more than the length of the tuples stored in the corresponding data structure.)

Because of the close relationship between the form of each data structure and the form of the corresponding database table, specifying the type of each column in the database table implies the form of the data structure, including the types of the values stored in the data structure. Therefore, the data specification file does not need to explicitly indicate the form of each data structure.

For ease of parsing, the prototype implementation of NERO expects the data specification file to contain a JSON object, where each key-value pair corresponds to a single data structure-database table pair. Each key is the name of the data structure, and the corresponding value is a list containing three entries: a “template,” a list of database column names, and the name of the corresponding database table:

- If the data structure is a list, the template is a JSON list containing a single item. This item is a JSON list representing a tuple, as described below.

- If the data structure is instead a dict, the template is a JSON object containing a single key-value pair with key “str”\(^3\) and value a JSON list representing a tuple, as described below.

The “tuple” in the template contains entries which are either “str” or “int” depending on the type of the corresponding tuple entries in the seed program data structure, or equivalently, the type of the corresponding column in the regenerated program database table. The length of the “tuple” in the template is the length of the tuples in the corresponding data structure, and one less than the number of columns in the corresponding database table (as described above, the database table has an additional column corresponding to a list index or dict key).

\(^2\)Because of a quirk in JSON, which is used by the fill interface to specify data structure contents, NERO requires dict keys, and therefore primary keys of tables corresponding to dicts, to be of string type. This restriction is straightforward to remove, however.

\(^3\)Again, to accommodate a quirk of JSON, NERO requires all dict keys to be strings and not integers.
As with the command specification file, it would be straightforward to modify the Nero implementation to accept a more human-readable or human-writable data specification file instead.

Figure 2-3 presents the data specification file for the student registration application:

- **student** is the name of the first data structure. It is a list of tuples containing two strings, as indicated by the template `[["str", "str"]]. This list will be regenerated to a database table containing three columns: one for the student’s id, and two for first_name and last_name, respectively. Observe that although the student data structure contains tuples of two elements, there are three columns in the corresponding database table because the list index is also regenerated into a database column (in this case id), which will be an integer primary key, whereas the first and last name columns correspond to the two string tuple entries. The name of the regenerated database table will be Student.

- **course** is the name of the second data structure. It is a dict, where keys are strings and values are tuples containing a string and an int, as indicated by the template `{"str": ["str", "int"]}. This dict will be regenerated to a database table containing three columns: one for the course number course_num (e.g. "6.033"), one for the name of the course, and one for the teacher_id. Here, observe that the dict key is regenerated into a column (in this case course_num) which will be a string primary key. As was the case before, the two tuple entries are also regenerated into their respective columns, a string column name and an int column teacher_id, whose types are implied by the types of the corresponding tuple entries. The name of the regenerated database table will be Course.

- **registration** is the name of the third data structure. It is defined in a manner similar to student, so for brevity I will avoid going into detail on its structure.

---

4The column teacher_id could refer to, for example, the ID of the teacher as stored in a Teacher database. However, none of the commands in the seed program access such a database; therefore, it is unnecessary to have a teacher data structure in the seed program corresponding to the Teacher database. Instead, Nero just treats teacher_id like any generic integer field.
Figure 2-3: Data specification file for the student registration application. This file is given in the format of a JSON object. Each key-value pair of the object corresponds to a single data structure in the seed program and corresponding database table in the regenerated program. The key in each key-value pair (e.g. student) is the name of the data structure, and the value is a list containing three entries: a “template” (e.g. 
[
  ["str", "str"],
  ["id", "first_name", "last_name"],
  "Student"
]
), a list of database column names (e.g. ["id", "first_name", "last_name"]), and the name of the corresponding database table (e.g. Student). If the data structure is a list (e.g. with student), the template is a JSON list containing a single item. This item is a JSON list (representing a tuple), with entries which are either “str” or “int” depending on whether the entry, and corresponding database column, should be a string or integer. If the data structure is instead a dict, the template is a JSON object (e.g. 
{
  ["str": ["str", "int"]},
  ["course_num", "name", "teacher_id"],
  "Course"
}
) containing a single key-value pair with key “str” (because, to accommodate a quirk of JSON, NERO requires all dict keys to be strings and not integers, though this restriction is straightforward to remove). The corresponding value is a JSON list representing a tuple in the manner specified above for list data structures.

Although the seed program does not actually use the list index of registration in any manner, NERO still regenerates a database table with the list index as an integer primary key, and the user is still required to specify a name for this column. However, it is straightforward to optimize out such unused columns if the user wishes, in the same manner that compilers can remove unused variables.

It is noteworthy that NERO is able to infer the functionality of the two commands in the student registration application despite the absence of commands which populate student and course information. The two commands can be inferred because NERO uses the fill commands it generates to populate data structures with the values required
Figure 2-4: Sample data for the student registration application example. The sample data includes three students and three students. Alyssa (ID 0) is signed up for all three courses. Ben (ID 1) is signed up for 6.857 only. Cy (ID 2) is not signed up for any course.

for the inference algorithm, and does not rely on the commands in the seed program to populate the data structures.

2.1.4 Sample Data and Execution

In this section, I present sample data, a sample set of executable commands, and the resulting execution, for the student registration application presented in Figure 2-1. Figure 2-4 presents sample data for the data structures used in the seed program: the student list, the course dict, and the registration list. The sample data matches the format indicated by the data specification file presented in Figure 2-3.

Next, Figure 2-5 presents a sample set of commands the user can execute on the seed program to explore the seed program’s functionality. Because the seed program is simply a collection of Python function definitions, with no entry point and no way to parse user input, the seed program is not immediately executable. As described by Section 2.1.2, NERO automatically generates a command loop, which reads commands from stdin and dispatches them to the Python functions in the seed program. As described by Section 2.1.3, NERO additionally generates a set of fill command handlers,
which read data from a specified file and populate data structures with the data in the file. When the seed program is augmented with the command loop and fill command handlers, it becomes executable. NERO automatically performs these augmentations to allow the inference algorithm to execute the seed program, but the prototype NERO implementation also makes the augmented seed program available to the user in case the user wishes to interact with it.

After augmenting the seed program with the command loop and fill command handlers, each of the sample commands in Figure 2-5 is executed in turn. These sample commands demonstrate how the augmented seed program executes. Note that the NERO inference algorithm issues similar commands, but the commands issued by the NERO inference algorithm use randomly generated inputs, and the NERO inference algorithm repopulates all data structures with newly generated random data using the fill commands before executing each command. This example first uses fill commands to populate data structures with the user-provided sample data presented

---

Figure 2-5: Sample sequence of executed commands for the student registration application example. A user could issue this sequence of commands to the seed program to explore the program’s functionality (with a caveat explained in the text). The first three commands are fill commands (Section 2.1.3); NERO automatically generates fill command handlers, which read data in JSON format from a specified file and load the data into the corresponding data structure. The three fill commands populate the data structures with the sample data presented in Figure 2-4. After the data structures are populated, the courses for each student are listed in turn. Then, students 2 and 1 (Cy and Ben) are each signed up for a class, and each students’ courses are listed again.

---

5 Repopulating every data structure allows the NERO inference algorithm to isolate the effects of each command.
Figure 2-6: Sample execution result for the student registration application example. The seed program (Figure 2-1) was augmented with a command loop (Section 2.1.2) and fill command handlers (Section 2.1.3). Then, the seed program was executed, and each command in Figure 2-5 was typed into the seed program. This figure presents the resulting interaction. Before each command, the command loop prompts the user for an input. When this sequence of commands is executed, the data structures are first populated with data using the fill command handlers. Then, each student’s courses are listed in turn: Alyssa is registered for three classes, Ben is registered for two, and Cy is registered for none. Cy and Ben then each register for a new course. Each students’ courses are listed again; the newly registered courses are included in the output for Ben and Cy.
in Figure 2-4. Next, each student’s courses are listed in turn. Then, two students are registered for new courses. Finally, each student’s courses are listed again. The resulting interaction in the command-line interface, including prompts for commands, user-issued commands, and the seed program output, is presented in Figure 2-6.

The augmented seed program is able to successfully execute the sample commands in Figure 2-5. Before each command is entered, the command loop prompts the user for the next command. The command is parsed and dispatched to the corresponding Python function. Any output is printed back to the console. The fill commands have no output. The list_student_courses command prints the name of the student, then information on each of their courses. Alyssa has three courses, Ben has one, and Cy has none. The register command prints the name of the student, then registers the student for the course. When each student’s course information is listed again, the newly registered courses are included, showing that the register command registered each student successfully.

2.1.5 Corresponding Regenerated Program

When Nero is provided with the example seed program, command specification file, and data specification file above, it executes carefully-chosen commands to infer the functionality of the seed program and outputs a regenerated program implementing the same functionality. In this section, I first present the regenerated program. I will present the inference and regeneration algorithms in Sections 2.3, 2.4.

Figure 2-7 presents the database schema regenerated by Nero for the student registration application. The regenerated database tables correspond to the three data structures used in the seed program. The algorithm for translating data structures to databases is straightforward because all the necessary information is provided by the data specification file, as explained above in Section 2.1.3.

Figure 2-8 presents one of the regenerated commands, list_student_courses, regenerated as the do_list_student_courses Python function. The command is regenerated in a manner similar to how Nero represents the seed program’s functionality, which is as a series of data structure accesses with interspersed print statements.
CREATE TABLE IF NOT EXISTS Student (id INTEGER PRIMARY KEY, first_name TEXT, last_name TEXT)
CREATE TABLE IF NOT EXISTS Course (course_num TEXT PRIMARY KEY, name TEXT, teacher_id INT)
CREATE TABLE IF NOT EXISTS Registration (reg_id INTEGER PRIMARY KEY, student_id INT, course_num TEXT)

Figure 2-7: Regenerated database schema for the student registration application. This figure presents the SQLite code used to define the tables in the regenerated program. Each line represents one of the three database tables. The first column in each table is either an INTEGER PRIMARY KEY, corresponding to a list index, or a TEXT PRIMARY KEY, corresponding to a dict key. The remaining columns have types corresponding to tuple entries as given in the data specification file (Figure 2-3). All column names are also specified in the data specification file.

Correspondingly, the regenerated command is implemented as a sequence of SQL queries (corresponding to data structure accesses in the seed program) and print statements, where intermediate results are stored and retrieved using a “context” list. The regenerated command also includes error-handling code which detects when a requested row was not found in the database table and gracefully aborts the command. Each data structure access adds an item to the context. Some accesses (including print statements) have no return value or have been optimized away, leaving a None value to be added to the context to avoid needing to renumber context references (though doing so is straightforward). Loops in the seed program are left as loops in the regenerated program, but instead of iterating over the data structure, a SELECT query is used to retrieve data, then the loop iterates over the result of the SELECT query. At the end of loops, the last several items in the context must be removed to reset the context to the state at the beginning of each iteration.

The register command is regenerated in a similar fashion to how the command list_student_courses was regenerated, so it is not shown. The regenerated program additionally contains a substantial amount of boilerplate (not shown) to generate the database tables, implement the command loop, and implement fill commands. Note that it is straightforward to store intermediate values in variables instead or otherwise clean up the regenerated code for readability; inferring the correct names to use for these variables may be a little more challenging, but a simple and reasonable
def do_list_student_courses(args):
    context = [(a,) for a in args]
    query = 'SELECT Student.id - 1, Student.first_name, Student.last_name FROM
    Student WHERE Student.id - 1 = :index'
    params = {'index': context[0][0]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found.
        return
    context.append(res[1])
    print('''Student: {context_set_0}, {context_set_1}'''.format(context_set_1=\n        context[1][1], context_set_0=context[1][0]))
    context.append(None)
    context.append(None)
    query = 'SELECT Registration.reg_id - 1, Registration.student_id, Registration .course_num, Course.course_num, Course.name, Course.teacher_id FROM
    Registration INNER JOIN Course ON Course.course_num = Registration .course_num WHERE Registration.student_id = :lookup'
    params = {'lookup': context[0][0]}
    context.append(c.execute(query, params).fetchall())
    for i0, _ in enumerate(context[4]):
        context.append(None)
        context.append((context[4][i0][3], context[4][i0][4], context[4][i0][5]))
    query = 'SELECT Course.course_num, Course.name, Course.teacher_id FROM
    Course WHERE Course.course_num = :key'
    params = {'key': context[4][i0][2]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found.
        return
    context.append(res[1])
    print('''{context_set_0} {context_set_1} {context_set_2}'''.format(\n        context_set_2=context[7][1], context_set_1=context[7][0], context_set_0=\n        context[4][i0][2]))
    context.append(None)
    context = context[:4]
    context = context[:1]

Figure 2-8: Regenerated list_student_courses command for the student registration application. This figure presents one of the functions in the regenerated program, implementing the same functionality as the do_liststudentcourses Python function in the seed program (Figure 2-1). The command is regenerated in a manner similar to how NERO represents the seed program’s functionality: as a series of data structure accesses with interspersed print statements. The regenerated command is implemented as a sequence of SQL queries (corresponding to data structure accesses in the seed program) and print statements, some of which are in loops, where intermediate results are stored and retrieved using a “context” list. The regenerated command also includes input validation code which detects when a requested row does not exist in the database table and gracefully aborts the command. Each database table access adds an item to the context. Some accesses have been optimized away, leaving a None value to be added to the context to avoid needing to renumber context references (though doing so is straightforward). At the end of loops, the last several items in the context must be removed to reset the context to the state at the beginning of each iteration. The regenerated program additionally contains a substantial amount of boilerplate (not shown) to generate the database tables, implement the command loop, and implement fill commands.
algorithm is to systematically generate variable names based on database table and column names.

The regenerated program includes a command loop and fill command handlers compatible with those generated for the seed program, so it can be executed using the sample data and commands presented in Section 2.1.4. The result of the execution is the same as the execution result of the seed program on the same data. The match between the two execution results is the expected behavior, because the regenerated program implements the same functionality as the seed program, except using databases instead of data structures. The correspondence between the seed and regenerated program is discussed in further detail in Section 2.5.

2.2 Supported Seed Programs

NERO infers the functionality of seed programs which use a subset of Python syntax. The seed program defines lists and dictionaries as global data structures and implements commands as functions that access these data structures. Informally, seed programs are allowed to:

- Initialize global data structures at the beginning of the program in fixed order, for example by defining them at the top of the file.

- Each command may perform an arbitrary sequence of reads and writes on global lists and dictionaries using values passed in as command parameters or retrieved from prior reads.

- Loop through a global list or a dict. Within the loop, the program can perform an arbitrary sequence of reads and writes, or have nested loops, etc. The loop must not be empty.

- Conditionally loop through a global list or a dict, i.e. loop through a list or a dict and perform a sequence of actions (reads/writes, loops, etc.) on a condition based on the value retrieved at that iteration. The condition must be a set of
equality checks of the form: “[element of the data structure] == [value in the context].” That is, each condition may be “Does the value just retrieved equal this other value?” Conjunctions of these conditions are also supported.

- Output values using a print function. Namely, the program is allowed to print strings, where some parts of the string may be filled in by string or integer values. For example, print the string “The number is _” with the blank filled in with a value retrieved from a global data structure.

I will present the supported syntax in detail in Chapter 3. Note that the student registration application seed program in Figure 2-1 satisfies these requirements.

NERO models seed programs which follow this specification in the following manner:

- Each seed program is a sequence of function calls (data structure accesses or print statements), including loops or conditional loops. (These are the operations the seed program is permitted to perform.)

- Function calls are connected by a context, representing the set of values available to the seed program.

- The context is initialized as an empty list, except for containing the input parameters passed to the command.

- Each function call is made with input arguments taken from the context. The return value is then appended to the context.

The purpose of the inference algorithm, described next, is to fit this model to the seed program given, i.e., to determine the sequence of function calls in the seed program and how the input arguments to each are constructed. Figures 2-9,2-10 in the following sections present how NERO models the two commands in the student registration application.
2.3 Inference

I now present how NERO infers the functionality of the student registration application. At a high level, NERO populates the data structures used by the seed program with a small number of elements, then executes the seed program and observes the resulting execution trace captured by the instrumentation. The trace contains the sequence of data structure accesses and print statements which result, including the concrete values passed as function arguments and the concrete values returned from data structure accesses. The inference algorithm then fits the model described in Section 2.2 to the trace by tracking the set of concrete values available in the context and matching them with the concrete values used as function arguments in the trace. Over the course of inference, NERO will repopulate the data structures as needed with carefully selected values to distinguish ambiguous concrete values and to explore more hidden paths in the seed program.

To infer the functionality of the student registration application, NERO first preprocesses the seed program to make it executable, and then infers the two commands separately, as described in the next several sections.

2.3.1 Initialization and Preprocessing

NERO begins by augmenting the seed program as follows.

- Recall from Section 2.1.2 that NERO automatically generates a command loop based on the command specification file in Figure 2-2. This loop reads commands from standard input and dispatches calls to the specified Python functions that implement the commands.

- Recall from Section 2.1.3 that NERO automatically generates fill command handlers based on the data interface file in Figure 2-3. These handlers are used by NERO inference algorithm to populate the seed program data structures with chosen values. One handler is generated for each data structure. Each handler takes a file name as a parameter, then reads data from that file in JSON format,
overwriting the contents of the data structure with the data in the file.

- NERO augments the seed program with instrumentation allowing NERO to observe the sequence of data structure accesses during the execution.

After augmenting the seed program with a command loop, fill command handlers, and instrumentation, NERO needs to perform an additional preprocessing step so that the information reported by the instrumentation can be used, described next.

When the instrumentation reports the sequence of data structure accesses, it reports which data structure is accessed by its unique identifier (UID). The instrumentation reports the UID of each data structure because the instrumentation does not know the name of each data structure. When the seed program initializes global data structures at the beginning of the program, the instrumentation causes the seed program to initialize an instrumented list or dict instead of a native Python list or dict. A UID, is assigned to each instrumented data structure when it is initialized. UIDs are assigned in order of initialization (i.e., UID 1 is the first data structure to be initialized). Therefore, NERO requires seed programs to initialize data structures in a fixed order (as noted in Section 2.2), so that UIDs are consistent even if the seed program crashes and is rebooted. This consistency allows NERO to handle crashes in the seed program.

To make sense of the sequence of data structure accesses reported by the instrumentation, NERO needs to determine which data structures (given by named) are assigned which UIDs. The instrumentation reports only the UID of each data structure accessed. Thus, NERO needs to know the correspondences between data structure names and UIDs so it can understand which data structures the instrumentation is referring to.

The inference algorithm uses fill commands to determine the name-UID correspondences. Each fill command handler generated by NERO populates a data structure by accessing its global variable by name. Thus, after the seed program is initialized but before inferring commands, NERO runs each fill command (which specifies a data structure by name) with random data, and observes which UID the instrumentation reports as being filled with data.
After running each of the fill commands in the student registration application seed program, NERO reports:

- **Command fill_course fills UID2**
- **Command fill_student fills UID1**
- **Command fill_registration fills UID3**

indicating that course has UID 2, student has UID 1, and registration has UID 3.

Once the name-UID correspondences have been inferred, the preprocessing step is complete and NERO proceeds to infer each of the commands implemented by the seed program.

### 2.3.2 Inferring Data Sources for register:

Once NERO has finished preprocessing the seed program, it proceeds to infer the functionality of each command in turn. NERO is able to infer each command independently, as discussed in Chapter 6. In the student registration application example, the first command is `register`, as specified by the command specification file presented in Figure 2-2. The implementation of `register` is given in the `do_register` function in the seed program, presented in Figure 2-1.

#### Initial Execution and Crash Prevention

The inference algorithm’s first task in inferring the `register` command is to execute the command to completion, preventing crashes which may occur due to IndexErrors or KeyErrors in the seed program.

NERO uses the fill command handlers to populate data structures with a small number of elements, each unique. It then executes the `register` command with randomly selected input command parameters, for example, `student_id_query=4311270` and `course_num_query="0b6e8ba6"`. This initial execution immediately results in an IndexError in the seed program when it tries to access the `student` list with index 4311270, which is much longer than the list length.

The NERO instrumentation detects the IndexError and reports:
When an exception occurs due to a data structure access in the seed program, the instrumentation captures:

- The type of the data structure accessed, in this case a list
- The UID of the data structure accessed, in this case UID1
- The function call which caused the exception, in this case `__getitem__`
- Any arguments, as a tuple, that were passed to the function, in this case a tuple containing a single element 4311270
- Any keyword arguments, as a dict, that were passed to the function. In this case there were none,\(^6\) represented by an empty dict `{}`.

The error message says that there was an error accessing the `__getitem__` function on the list with UID 1 and that the index used for the `__getitem__` function was 4311270. Because the crash occurred on a `__getitem__` call, NERO infers the crash was due to an IndexError. NERO analyzes the execution to determine how to prevent this error from occurring in the future. It first reconstructs the context of the execution, which is the list of values available to the command at each point in the execution. The context is initialized with the command parameters:

```
0: (4311270,)
1: (0b6e8ba6,)
```

Each item in the context is a tuple. Each item is a tuple because typically data structure accesses return tuples if they return anything at all; for example, because data structures are expected to be lists or dicts containing tuples, a list index lookup or dict key lookup will return a tuple. When NERO reconstructs the context to determine the cause of the crash in register, the first item in the context is a single-element tuple containing the `student_id_query` parameter 4311270, and the second item contains the `course_num_query` parameter 0b6e8ba6.

\(^6\)In fact none of the data structure accesses currently supported by NERO use keyword arguments, but the instrumentation captures them in case it is needed in the future.
NERO determined previously during initialization that the list with UID 1, the list that was accessed causing a crash, is student. By using the context, NERO infers that the concrete input of \_\_getitem\_\_, 4311270, came from the first input parameter \texttt{student\_id\_query}. In future executions, NERO will enforce a constraint that parameter \texttt{student\_id\_query} must be less than the length of list \texttt{student}, to prevent crashing from this same error.

NERO executes the \texttt{register} command again, now observing the newly generated constraint, and verifies that the IndexError does not occur. This time, \texttt{register} executes to completion, so NERO is now able to infer the functionality implemented by the command.

\section*{Analyzing a Successful Execution}

Now that the \texttt{register} command is no longer crashing, the inference algorithm proceeds to infer the functionality implemented by the command. To do so, it executes the command again and observes the resulting sequence of function calls (data structure accesses, etc.) in the execution.

NERO populates the data structures again with a small number of elements. It executes the \texttt{register} command again, using new random inputs, for example, \texttt{student\_id\_query} = 9 and \texttt{course\_id\_query} = a5091eb0. This time, the value assigned to parameter \texttt{student\_id\_query} is a small number that respects the constraint imposed in the previous step. \texttt{register} now executes successfully and results in the following trace:

\begin{verbatim}
1 Call: (list) -1 -1 UID1 \_\_getitem\_\_: args[(9,)] kwargs[{}] -> (’aa0b9a8d’, ’15caе748’)
2 Call: (print) -1 -1 print print : args[( ’aa0b9a8d’, ’15caе748’)]
     kwargs[{}] -> None
3 Call: (list) -1 -1 UID3 append : args[([9, ’a5091eb0’]),] kwargs
     [{}] -> None
\end{verbatim}

This trace captures each function call (data structure access or print statement) on a separate line. For each data structure accessed, the instrumentation records in the
trace:

1. The type of the data structure: list, dict, or print

2. Iteration information, discussed in Subsection 2.3.3. In this trace, it is recorded as -1 -1 since there is no iteration.

3. The UID of the data structure, e.g. UID1, or print if it was a print statement

4. The function (accessor method) being called, e.g. __getitem__ (i.e., list index lookup), or print if it was a print statement

5. The arguments (args) of the function, given as a tuple, e.g. (9,) (a tuple with a single element)

6. The keyword arguments (kwargs) of the function, given as a dict. Keyword arguments are not actually used by any function supported by NERO (and therefore the keyword arguments are always an empty dict {}), but the instrumentation captures them in case they are needed later.

7. The return value of the function, e.g. the tuple (‘87f089cc’, ’5c2d3003’), or None if there was no value returned.

NERO reads this trace and learns that the command involves a list access operation (__getitem__ on line 1), a print operation (print() on line 2), and a list append operation (append() on line 3).

With this successful execution trace, NERO now starts to infer the source of each concrete value in the trace. To do so, it follows the same procedure used to infer the cause of the crash in Section 2.3.2: by reconstructing the execution context of values available to the seed program at each point in the execution. For each function call recorded in the trace, NERO searches for the concrete values of its input arguments in the context, and infers the source of these arguments from the resulting matches. After each function call in the trace, NERO updates the context with the return value. This process is shown below.

The initial context contains the command parameters:
The first entry in the context (line 0) is a single-element tuple containing the first command parameter, 9. The second entry (line 9) is a single-element tuple containing the second command parameter, 'a5091eb0'.

NERO analyzes the first function call in the trace to determine the source of its input argument, which was 9. NERO searches for this value in the context, and finds a single match: the first entry of the context (line 0), corresponding to the first command parameter. Thus, the algorithm infers that the source of the argument for the first function call of register is the first command parameter.

After inferring the source of the argument to the first function call, NERO then updates the context with the return value of the first function call:

0: (9,)
1: ('a5091eb0',)
2: ('aa0b9a8d', '15cae748')

Now NERO analyzes the second function call in the trace to determine the source of its input arguments by comparing the arguments to the update context. On line 2 of the trace, the function print() is invoked with input arguments ('aa0b9a8d', '15cae748'), where both strings come from line 2 of the context. Recall that line 2 of the context contains the return value of the first function call in the trace. NERO therefore infers that the second function call uses values returned by the first function call as arguments. The second function call, print(), has no return value, so NERO appends None to the context. On line 3 of the trace, the function append() is invoked with a two-element tuple as an input argument, (9, 'a5091eb0'). NERO searches for the two elements separately in the context, and finds they are the two command parameters.

As the inference result, NERO concludes that the register command (implemented in Figure 2-1) contains a print operation using the results of the access to list student and an append operation using the two input command parameters.

The inference algorithm outputs an intermediate representation (IR) of this behav-
ior, presented in Figure 2-9. This IR captures the sequence of data structure accesses made. For each data structure access, the IR indicates

- The type and UID of the data structure accessed, e.g. list:UID1, (or in the case of a print statement, print:print)

- The accessor method used (i.e., the function that was called), e.g. __getitem__ or print

- The arguments and keyword arguments\(^7\) (see below).

For each argument, the IR captures the set of possible locations in the context the argument could have come from. Each *context location* is represented by a pair, e.g. (0, 0) on line 1 of the example IR. Recall that the context is a list of tuples. The first number in the context location is the list index; in this case, 0 refers to the 0th element in the list, i.e., the first command parameter. The second number is the tuple index; in this case, 0, because the first command parameter is a single-element tuple. If an argument could possibly come from multiple locations in the context, this is represented in the IR as a set containing multiple context locations. NERO tries to resolve context locations unambiguously, but sometimes ambiguity is inherent in the seed program, e.g. if a list is accessed twice with the same index lookup (although no ambiguity occurs for this command). The argument for the first list lookup in register is unambiguous, therefore represented as a set containing a single context location, {(0, 0)}.

For a print statement, the IR additionally captures the print statement *template*. In line 2 of the IR, the template is '{context_set_0} {context_set_1}', i.e., a format string containing two placeholders separated by a space. The placeholders are filled by {(2, 0)}, {(2, 1)}], referring to one context location for each placeholder. A context location of the form (2, n) refers to the nth item of the tuple stored in the second context location. Because register has two parameters, the second context location is the return value of the first function call in the command, i.e., the tuple

\(^7\)Again, keyword arguments are currently unused
Figure 2-9: IR of the `register` command inferred by NERO. The first operation is a `__getitem__` (i.e., index lookup) call to the list with UID 1, where the index item is the context location (0,0) (i.e., the first command parameter). The second operation is a `print` statement, where two context items are printed separated by a space. The two context items are context locations (2,0) and (2,1) respectively; because there were two command parameters, (2, n) refers to the values returned by the list lookup (UID1 (the students list) is a list containing pairs of strings). The third operation is a `append` call to the list with UID 3, where the item to be appended is a pair constructed from context locations (0,0) and (1,0), i.e., the two command parameters.

retrieved from `student`. In other words, the IR indicates in line 2 that the second function call in `register` is a print statement, where the two values returned in line 1 are printed separated by a space.

Similarly, line 3 of the IR indicates a list append operation, where the input argument is a pair constructed from context locations (0,0) and (1,0), i.e., the two command parameters.

### 2.3.3 Inferring Loops for `list_student_courses`:

Once `register` has been inferred, the inference algorithm moves on to inferring `list_student_courses`. Similar to inferring command `register`, NERO begins by executing the `list_student_courses` command with random inputs and discovering constraints which prevent the program from crashing.

After these initial steps, an execution using input `student_id_query=7` produces the following trace.

```python
1 Call: (list) -1 -1 UID1 __getitem__: args[{(7,)}] kwargs[{}] -> ('ed0843b2', '0a71824c')
2 Call: (print) -1 -1 print print: args[('Student: ed0843b2, 0a71824c',)] kwargs[{}] -> None
3 Call: (list) -1 -1 UID3 __iter__: args[()] kwargs[{}] -> <__main__.InstrumentedIterator object at 0x7f6478077240>
```
As was the case with the register command above, the trace captures the type and UID of each data structure accessed, as well as each accessor method (i.e., the function which was called) and the arguments/keyword arguments and return value of these functions. However, in this trace, the seed program iterates over a data structure, so the instrumentation additionally captures iteration information, described next.

In Python, when a program uses a for loop to iterate over a data structure, the __iter__ function is called on the data structure (e.g., in line 3 of the above trace), which returns an iterator object. In this case, an instrumented version of the iterator is returned instead, as shown in line 3 of the trace. Then, the iterator’s __next__ function is repeatedly called, once at the beginning of each iteration, e.g., in lines 4 and 5 of the trace above. Calling the __next__ function repeatedly causes each of the data structure’s contents to be returned in turn.

When the __next__ function on the iterator is called, the instrumentation captures the type and UID of the parent data structure (in the case of lines 4 and 5, it is the list with UID 3). Additionally, it captures a UID for the iterator itself, which is reported as 0 on lines 4 and 5, as well as the index of the iteration, which starts at 0 and increments once for each iteration on lines 4 and 5. The iterator UID is 0 the first time __iter__ is called on each data structure, and increments by one each time thereafter. The iterator UID allows NERO to distinguish between consecutive or nested loops over the same data structure, e.g., if the seed program contains two nested loops over the same data structure, when the instrumentation reports that __next__ was called, the iterator UID allows NERO to determine whether the inner or outer loop began a new iteration. For functions other than __next__, the iterator ID and iteration index are both reported as −1, for example in lines 1–3 of the trace.

Similar to when tracking the data sources for command register, NERO analyzes
line 1 of the trace by reconstructing the execution context and infers that there is a list access operation using the input parameter student_id_query as the index. Line 2 is slightly different from that of command register: the print operation contains a string 'Student: ed0843b2, 0a71824c'. NERO searches for substrings which match values in the context, and finds that 'ed0843b2' and '0a71824c' match the values returned by the list access on line 1. NERO therefore infers the print operation uses the template string 'student: {}, {}' where each {} denotes a placeholder, filled by the two values returned by the list access on line 1.

Next, line 3 of the trace calls the __iter__() function on a list with UID 3, which represents the start of a loop over the list. Lines 4, 5, etc. each call the __next__() function on the same list, each representing the start of a loop iteration. Because the UID 3 corresponds to the list registration, NERO infers that the seed program command has a loop that iterates over registration.

Then, NERO notes that nothing happens between any of the calls to __next__, that is, the loop body was never executed. Because NERO assumes that any loop has a non-empty loop body, NERO infers that the observed loop over list registration must contain a loop body that executes only when certain unidentified conditions satisfy.

NERO then tries to discover the unknown condition for entering the loop body. NERO executes the seed program multiple times, each time assigning carefully chosen values to input parameters and data structures with the constraint that a potential condition would hold. For example, NERO considers the condition where the second value returned by the loop over registration (e.g. '5b0e0c4b' in line 4) should match the first value retrieved from the student list (e.g. 'ed0843b2' in line 1). In this case, NERO would impose the constraint that the first column of student should be a subset of the second column of registration, i.e., that for each tuple stored in the student list, the first entry should match the second entry of some tuple stored in the registration list. Thus, when the seed program iterates over registration, regardless of which entry in the student list was retrieved in the first step of the trace,

---

8Technically, __next__ is called on the iterator returned from line 3, not the list itself.
the condition will satisfy at least once, assuming NERO guessed correctly that the second value returned by the loop over registration should match the first value retrieved from the student list.

When NERO executes the command enforcing the constraint that the input parameter student_id_query is equal to some element in the first column of list registration, the seed program produces the following trace:

1 ... (same as the previous trace, omitted)
2 Call: (list) 2 5 UID3 __next__: args[()] kwargs[{}] -> (6, '104f54ef')
3 Call: (dict) -1 -1 UID2 __iter__: args[()] kwargs[{}] -> <__main__.InstrumentedIterator object at 0x7f6478077470>
4 Call: (dict) 0 0 UID2 __next__: args[()] kwargs[{}] -> 'bd4c5a00',
5 ... (9 more iterations on dict of UID2, omitted)
6 Call: (list) 2 6 UID3 __next__: args[()] kwargs[{}] -> (7, '49dce070')
7 ... (3 more iterations on list of UID3, omitted)

The first few lines are conceptually similar to the previous trace that iterates over a list of UID 3. However, on line 3 of this new trace, there is a new function call __iter__() to a dict of UID 2. Lines 3 through 5 continues to loop over this dict of UID 2. Then, line 6 returns to the loop over the list of UID 3. From this information, NERO infers that the seed program command contains a loop over registration whose loop body executes only when the input parameter student_id_query equals the first value retrieved from list registration. When this condition satisfies, the command executes a nested loop that iterates over the dict of UID 2, which is the dict course.

NERO again notes that the trace does not contain anything between different iterations over course and infers that there is a conditional block within the inner loop. NERO performs the same steps to infer the condition inside this loop.

Note that NERO is able to perform exactly the same procedure to infer the condition for the inner loop as the outer loop, although the two conditions are implemented
using different syntax (Figure 2-1). The outer loop is implemented with a `continue` statement to skip the body when equality does not hold, whereas the inner loop executes the body in a `then` clause when equality does hold. Because these two implementations are logically equivalent and lead to the same program behavior, `NERO` is able to infer both conditions in the same fashion, even though `NERO`'s domain-specific language does not make any mention of `continue` statements. This flexibility in the implementation of the seed program is an advantage of `NERO`'s dynamic inference algorithm compared to a static analysis of the seed program code.

As the inference result, `NERO` concludes that the `list_student_courses` command contains two nested loops that both have conditional checks before entering the loop body. The inner loop prints the course information using values retrieved from data structures.

The intermediate representation returned by the inference algorithm is presented in Figure 2-10. As was the case with `register`, the IR captures the sequence of data structure accesses/print statements and how their arguments are constructed. This example additionally shows how the IR represents loops. Each loop contains a `__iter__` statement as the “head,” plus a “body” which is a sequence of data structure accesses and print statements, where the first access is always a `__next__` operation. The IR additionally captures any conditions for conditional loops, e.g. `((1, (0, 0)),)`. The captured condition `((1, (0, 0)),)` is a single-element tuple,\(^9\) which means the condition only features a single comparison (as opposed to a conjunction of comparisons). In this case, the first column of the data structure being iterated over (indicated by the `1`\(^{10}\)) is compared to the value in context location `(0, 0)`, i.e., the command parameter. As required by `NERO`’s domain-specific language, the comparison is required to check for equality, i.e., the loop executes if and only if the two values compared are equal.

---

\(^9\) The single element is the pair `(1, (0, 0))`.  
\(^{10}\) The `1` indicates the first element of the tuple from the list. A `0` would refer to the list index.
Figure 2-10: IR of the `list_student_courses` command inferred by Nero. The first step is a list index lookup using the command parameter. The second is a print statement printing out the result of the first step. The third is a conditional iteration over a data structure. In this loop is a nested conditional iteration over another data structure. In the inner loop is a dict key lookup followed by a print statement.

### 2.3.4 Ambiguity in the Inferred Representation

When inferring the functionality of a seed program, Nero must infer which context location contains the value used for each input argument of each function call (data structure access or print statement) in the seed program. Nero infers which context location contains the value used for each input argument by matching the concrete value stored in each context location against the concrete value used for the input argument. However, sometimes there are multiple context locations containing a matching value. In this thesis, I call this situation an ambiguity for a function call input argument.

When an ambiguity occurs for a function call input argument, Nero tries to determine which context location is actually used by the seed program as the source of the input argument. Nero tries to determine which context location is used by the seed program by repeatedly executing the seed program with different inputs and data structure contents until the ambiguity is resolved.

In some cases, ambiguity necessarily occurs in the seed program. For example, if the same list is accessed twice using the same index, both accesses will return the same value. Alternatively, inside a conditional loop which compares the value in one context

---

11Because each context location corresponds to a variable in the seed program, one might also call it a variable ambiguity.
location $C_1$ against a value in another context location $C_2$, $C_1$ and $c_2$ will necessarily contain the same value inside the loop body. I call this situation where ambiguity necessarily occurs a true ambiguity. (An ambiguity other than a true ambiguity is a false ambiguity.) In the event of a true ambiguity between two context locations $C_1$ and $C_2$ for an import argument of a function call $F$, it is impossible for NERO to determine which of $C_1$ and $C_2$ is used by the seed program as the source for the input argument of $F$. NERO cannot distinguish between $C_1$ and $C_2$ because using either $C_1$ or $C_2$ as the source of the input argument of $F$ will lead to a semantically equivalent program. However, due to this semantic equivalence, it is not necessary for NERO to determine which context location is used; NERO can choose either $C_1$ or $C_2$ to use arbitrarily in the regenerated program.

Although it does not matter which context location NERO selects in the event of a true ambiguity, NERO needs to detect the true ambiguity to avoid repeatedly executing the seed program forever. It is theoretically possible to analyze the inferred program to detect the true ambiguity, but if the program is complex, this analysis may be difficult to perform. NERO takes the simpler approach of assuming the ambiguity is true if, after several repeated executions, the ambiguity has not been resolved. For more details on the ambiguity resolution algorithm, see Section 6.6.2. As I discuss in Section 6.8.5, NERO is able to resolve all false ambiguity with high probability, which is sufficiently good for practical purposes and justifies the use of the simpler approach instead of analytically determining whether the ambiguity is true.

The IR inferred for `list_student_courses` (Figure 2-10) contains an ambiguity in lines 7 and 8, expressed by a set containing two context locations, $\{(4, 2), (6, 0)\}$. The location $(4, 2)$ refers to the second item in the tuple retrieved from iterating over `registration` (UID 3) in line 3–4, and $(6, 0)$ refers to the dict key retrieved from iterating over `course` (UID 2) in line 5–6. This ambiguity is an example of a true ambiguity, an ambiguity inherent in the seed program. The ambiguity is true because lines 7 and 8 occur inside a conditional loop body (line 5) where the retrieved

---

12The iterator is retrieved in line 3, but `__next__` on line 4 is the accessor method which actually returns the contents of the data structure.
element 0 (i.e., the dict key of course) is checked for equality against context location (4, 2) (i.e., second item retrieved from registration). Therefore, on lines 7 and 8, these two values are always equal. During the inference algorithm, NERO notices this ambiguity, and executes the seed program several times with different inputs. Because the inputs used vary widely, the probability this ambiguity is coincidental is low, so after several executions NERO concludes the ambiguity is a true ambiguity and moves on. NERO arbitrarily selects one of the two ambiguous context locations to use in the regenerated program.

Next, I discuss how NERO constructs the regenerated program for the student registration application.

### 2.4 Regeneration

Once NERO has inferred the model of each command, it regenerates a database program written in Python with SQLite, which implements each command as a function as well as a command loop and the fill interface, which can be used to populate the database with arbitrary values. Boilerplate code to generate the tables is also included.

NERO first determines the database schema for the regenerated program, presented previously in Figure 2-7. The schema is constructed in a straightforward manner, explained previously in Section 2.1.3, based on the data specification file, presented previously in Figure 2-3.

To regenerate the commands, NERO begins with the IRs returned by the inference algorithm, previously presented in Figures 2-9,2-10. The regeneration algorithm is straightforward, and the regenerated code closely follows the IR inferred by the inference algorithm. First, the inference IR is converted to a second intermediate representation (the SQL IR, explained below), by preserving the nested structure of the inference IR but converting each step in a 1:1 manner to a SQL query:

- A list index lookup or dict key lookup is replaced by a SELECT query conditioning on the primary key of the corresponding table.
• A list index store is replaced by a UPDATE query conditioning on the primary key of the corresponding table

• A list append is replaced by a INSERT INTO query

• A dict key store is replaced by a INSERT OR REPLACE INTO query

• A print statement is left as a print statement

• For loops, recall that a loop in the IR has two parts: a head (___iter___) and body (sequence of commands starting with ___next__). The head is replaced by a SELECT query which retrieves all rows of the corresponding table, or if the loop has a condition, retrieves all rows matching the condition. The ___next___ in the body is skipped, and the rest of the body is placed in a for loop iterating over the return value of the head, and converted to the SQL IR in a recursive manner.

After the SQL IR has been generated, NERO performs some optimization on it. In the prototype implementation of NERO, the main optimization performed is simplifying loops. If a loop contains a nested conditional inner loop, NERO will try to combine the head of the inner loop with the head of the outer loop using a JOIN query as appropriate. For example, in list_student_courses, the inner loop’s condition compares the primary key of Course (course_num) with the third column of Registration (course_num), therefore NERO generates the following JOIN operation:

```sql
SELECT Registration.reg_id - 1, Registration.student_id,
       Registration.course_num, Course.course_num, Course.name, Course.teacher_id
FROM Registration INNER JOIN Course ON Course.course_num = Registration.course_num
WHERE Registration.student_id = :lookup
```

In this SQL query, strings :lookup0 and :lookup1 are placeholder values for which the correct values are substituted by the SQLite library. The - 1 offsets are an artifact to make the regenerated program compatible with the seed program, since SQL ids are 1-indexed whereas Python lists are 0-indexed, and can be optimized away.
The table and column names are provided in the data structure specification file in Figure 2-3.

Once this JOIN operation has been performed, NERO simplifies the SQL IR by replacing the nested loop with a single loop over the joined query’s return value. The body of the new loop is the original body of the inner loop, plus some boilerplate to ensure context locations remain intact.\textsuperscript{13} (If the outer loop contains additional operations before or after the inner loop, this optimization procedure cannot be performed.)

After the SQL IR has been optimized, NERO converts it into Python code using SQLite databases, thereby regenerating Python functions implementing the commands in the seed program. The regenerated Python function operates by performing SQL queries and storing intermediate results in a context list, following the SQL IR. Future work can be done to improve the readability of the regenerated code; for example, variables with systematically generated names based on database and column names can be used instead of a context list.

NERO additionally generates boilerplate code for creating the database tables, as well as a command loop and code implementing the fill commands used during the inference algorithm.

The regenerated \texttt{list\_student\_courses} command was presented previously in Figure 2-8. The full regenerated program is available in the Appendix (Figure A-12).

\section{Sample Behavior}

Section 2.1.4 provided sample data for the student registration application, and presented the behavior of the seed program on this data. In this section, I discuss the behavior of the regenerated program on the same data.

The regenerated program differs from the seed program in how it stores data, but the regenerated program otherwise implements the same functionality as the seed program. The regenerated program uses a database to store data, while the seed

\textsuperscript{13}These can be optimized away.
program uses Python data structures. As discussed below, the two programs produce the same behavior on the sample data, except data in the regenerated program persists across multiple executions.

The regenerated program includes a command loop and fill command handlers compatible with those generated for the seed program (Sections 2.1.2, 2.1.3). Therefore, the sample data and commands presented in Section 2.1.4 can be used for the regenerated program as well.

When the sample commands in Figure 2-5 are executed in the regenerated program, the fill commands load the data in Figures 2-4 into the regenerated program database. Then, the remaining commands execute. The resulting output is the same as the output for the seed program, presented in Figure 2-6. The match between the two outputs is the desired behavior, because the regenerated program should reproduce the functionality of the seed program.

However, there is one substantial difference between the behavior of the seed program and the regenerated program. The regenerated program is a database program, and the seed program is not. Therefore, data stored in the seed program is erased if the program exits, but data in the regenerated program is saved in the database. (The persistence of data is, after all, the purpose of writing a database program.) Therefore, when executing the sample commands on the regenerated program, the user could, between any two commands, exit the program, then come back later and resume the execution. The resulting behavior will be the same as if the user had never exited the program at all. By using Nero, the user is able to utilize the advantages of databases without needing to know how to implement a database program themselves.

2.6 Example Summary

The student registration application example illustrates how a user can use Python data structures to mimic databases in a seed program, and how Nero infers the functionality of the seed program and regenerates a program using SQLite but implementing the
same functionality. A user can use a list of tuples to store information canonically associated with an integer ID, such as with the student list, which is a common practice when using databases. A dict is appropriate when information is canonically associated with a string, such as a course number with the course list. A list can also be appropriate when information has no canonical identifier at all, such as with the registration list, because a list makes it easy to append new values.

A user can use NERO to generate a database program, including all of the necessary boilerplate, by implementing a simple and easy-to-write Python program. Although in the case of Python SQLite the boilerplate is relatively simple, a user may instead wish to regenerate to a more complex target, whether another language or a more complex (e.g., distributed) database, which may be much more difficult or tedious to write by hand. Though the prototype implementation of NERO only regenerates to Python SQLite, it is straightforward to write a new regeneration module to regenerate to a different target instead.

This example illustrates NERO’s ability to infer the functionality of the seed program by using instrumentation to capture much of the execution structure: each data structure access is recorded in detail, lending feasibility to the inference task. Despite the linear format of the trace reported by the instrumentation, NERO is able to infer the behavior of seed programs which iterate over data structures because the instrumentation captures the start of each individual iteration. As demonstrated in list_student_courses, inference can be done with nested loops as well, because NERO can infer the contents of a loop body in the same way it infers any sequence of data structure accesses.

Seed programs with aspects not directly captured by the instrumentation, such as conditional statements, begin to pose a challenge to NERO’s inference algorithm. Because the instrumentation does not directly indicate the presence of a conditional, nor what values are being compared, NERO is only able to infer conditionals by imposing strong assumptions on the presence of conditionals. Namely, conditionals are required to occur only at the start of a loop body, so that NERO can infer a conditional did not execute when the loop iteration gives an empty trace, and conditionals
are required to be equality checks, so NERO can enumerate all possible conditions and prune them down through repeated executions. The conditionals which seed programs are permitted to use allow NERO to impose constraints to cause conditions to execute, allowing NERO to infer the condition, as shown in list_student_courses (Section 2.3.3)

This example also illustrates NERO’s ability to regenerate an efficient program which implements the same functionality as the seed program. The list_student_courses command’s seed program implementation contained an inefficient nested loop which iterated over the registration and course data structures to retrieve the courses which the requested student is registered for. In the regenerated program, NERO was able to optimize the nested loop into a single loop over the result of a SELECT query which performs a JOIN operation on the database tables corresponding to the two data structures accessed by the nested loop in the seed program. Because NERO infers an intermediate representation capturing the core functionality of the seed program, it is able to perform optimizations which are designed to increase the efficiency of the regenerated program, such as the JOIN operation presented in this example.

NERO takes an approach where it assumes the seed program is equivalent to some program written in its domain-specific language, then executes the seed program and records enough information through instrumentation that it can efficiently filter down possible programs which are equivalent to the seed program. This approach allowed NERO to infer the functionality of the seed program where it had a non-semantic difference with the domain-specific language, namely the use of the continue statement in the outer loop of the do_liststudentcources Python function (Figure 2-1). NERO regenerated a SELECT query with a WHERE clause for the outer loop of do_liststudentcources. This query is the same as what would have been regenerated if the outer loop of do_liststudentcources had used the equivalent construction in the domain-specific language (like with the inner loop of do_liststudentcources). This type of flexibility is not possible with a static analysis approach, which can only infer programs explicitly accommodated by their domain-specific language.
Chapter 3

NERO Domain Specific Language

NERO supports seed programs which define a set of global list and dict data structures and implement a set of commands as Python functions which access the global list and dict data structures. Each list or dict should store tuples of string or int values, and is meant to mimic a database table (see Section 4.2). Commands can perform an arbitrary sequence of reads and writes on the global data structures. Commands are also allowed to loop over data structures. Commands are expected to output values by calling the print function.

I use a domain-specific language (DSL) to characterize the class of seed programs supported by NERO. NERO supports seed programs which conform exactly to this DSL, as well as seed programs which are equivalent (in terms of the sequence of global data structure accesses and print statements resulting in each execution) to some program conforming to the DSL. Allowing equivalent programs provides seed programs with syntactic flexibility.

I first present the grammar of the NERO DSL in Section 3.1. I discuss the scope of programs supported by the DSL in Section 3.2. I then discuss the syntactic flexibility provided to seed programs in Section 3.3.
Program := List* Dict* Command*
List := l = empty list \( l \in \text{ListVar} \)
Dict := d = empty dict \( d \in \text{DictVar} \)
Command := def f v*: Step*
Step := print v* | l append v* \( v \in \text{Var} \)
| l[i] = v* | v* = l[i] \( i \in \text{IntVar} \)
| d[s] = v* | v* = d[s] \( s \in \text{StrVar} \)
| Iterate: Body
Iterate := for a+ in l \( a \in \text{LoopVar} \)
| for a, a+ in enumerate l \( \text{Var} = \text{IntVar} \cup \text{StrVar} \)
| for a in d
Body := Step+ | if Cond+: Step+ \( \text{LoopVar} \subset \text{Var} \)
Cond := a == v

Figure 3-1: Abstract syntax for seed programs. Nonterminal symbols are defined on the left. Each lowercase letter \( (l, d, \ldots) \) represents a terminal symbol which represents a variable. Variables fall into different categories, indicated on the right, and defined in Section 3.1.1. Variables must obey additional restrictions defined in Section 3.1.3. The use of \( + \) in \( v^+ \) and \( a^+ \) indicate that tuples of variables should be used, because lists and dicts store tuples of values. Cond\(^+\) indicates that a conjunction of Conds can be used in the if-statement.

### 3.1 Grammar of the NERO DSL

The grammar for the domain-specific language of seed programs supported by NERO is presented in Figure 3-1. I explain the grammar in this section.

Terminal symbols in the grammar which represent variables fall under a number of overlapping categories, which I explain in Section 3.1.1. I explain the nonterminal symbols in the grammar in Section 3.1.2. The variables used in the NERO DSL must follow an additional set of restrictions, which I explain in Section 3.1.3.

#### 3.1.1 Variable Categories

Variables used in NERO seed programs fall into different overlapping categories, which I explain here. Lowercase letters \( (l, d, f, v, i, s, \text{and } a) \) in the syntax are terminal symbols which represent variables. Each letter denotes a variable belonging to a different category. The categories are as follows:

- **ListVar**: A global list data structure.
• **DictVar**: A global dict data structure.

• **Name**: The name of a Python function which implements the command.

• **Var**: A variable storing a single integer or string value used by the command.

• **IntVar**: A variable storing a single integer value used by the command. Note that IntVar is a subcategory of Var.

• **StrVar**: A variable storing a single string value used by the command. Note that StrVar is a subcategory of Var.

• **LoopVar**: A variable storing a single integer or string value retrieved by looping over a list or dict. Note that LoopVar is a subcategory of Var.

Any Var is either an IntVar or a StrVar. LoopVars may be either IntVars or StrVars.

Variables can only be used in a location allowed by the categories they are in. For example, suppose `foo` is a command parameter of integer type. Then `foo` is a Var, so it can be used where `v` appears in the syntax. It is also an IntVar, so `foo` can be used where `i` appears in the syntax. It is neither a StrVar nor a LoopVar, so `foo` cannot be used where `s` and `a` appear in the syntax. Variables in seed programs must additionally obey restrictions defined in Section 3.1.3.

Having defined the categories of variables, I next define nonterminal symbols used in the syntax.

### 3.1.2 Nonterminal Symbols

I now explain each nonterminal symbol used in the syntax.

• **Program**: A seed program. Seed programs begin with a sequence of declarations for global data structures. Each data structure is either a Python list (List) or a Python dict (Dict). The seed program implements a list of commands (Command).
Observe that the order in which global data structures are initialized when the seed program is executed is determined by the order in which they are declared. Therefore, data structures are initialized in the same order each time the seed program is loaded and executed. The fixed initialization order is important for reasons discussed in Section 5.1.

- **List**: A declaration of a global list data structure. Global list data structures store fixed-length tuples of strings and/or integers.

- **Dict**: A declaration of a global dict data structure. Global dict data structures have string keys,\(^1\) and values are fixed-length tuples of strings and/or integers.

Global data structures are initialized empty in the seed program, but NERO will populate them before executing commands. See Section 4.2 for more details on the form of global data structures, as well as how NERO populates global data structures.

- **Command**: A Python function implementing a command. Each command has a name \(f\), a list of input parameter variables \(v\), and a sequence of instructions (Step).

The seed program is not by itself executable, because it only defines a collection of functions and does not include any way to input commands from the user. NERO will use a specification file written by the user to automatically augment the seed program with a command loop, which reads commands from stdin and dispatches them to the appropriate Python functions (see Section 4.1). The command loop allows the inference algorithm to execute the seed program.

- **Step**: An instruction in a command. A Step is allowed to be a data structure access or a print statement, explained in more detail below. Each Step can be:
  
  - A print statement that uses the Python `print` function to display Vars to the standard output. (Not shown in the grammar: a print statement is

\(^1\)Due to a quirk in JSON, as explained in a footnote in Section 4.2, NERO requires dict keys to be strings.
allowed to use a Python format string as a template. In other words, the string printed can be a template with placeholders filled by Vars.)

- An append statement that uses the Python append function to add a tuple of Vars to the end of a list

- A write statement that overwrites a list/dict element with a tuple of Vars

- A read statement that retrieves a list/dict element and stores it into a tuple of Vars. Vars used to store values retrieved from read statements are subject to the Single Assignment restriction defined in Section 3.1.3.

- A loop that iterates over a list/dict (Iterate) and performs a nonempty sequence of operations (Body) for each iteration.

Throughout this thesis, I use the term function call to refer to a Step in the seed program, because in Python, each step is either a function call to a method of a data structure or a call to the print function.

- **Iterate**: The “head” of a loop, which iterates over a list/dict. The loop over a data structure allows the seed program to access each element in turn. A loop over a list retrieves each tuple in the list, and stores it into a tuple of variables (LoopVar\(^+\)). A loop over a list is allowed to access the Python enumerate function, which additionally gives access to the list index of each tuple, stored into a LoopVar. A loop over a dict retrieves each key in the dict, and stores it into a LoopVar. LoopVars are subject to the Scoping restriction defined in Section 3.1.3.

- **Body**: The body of a loop. A Body may contain any non-empty sequence of Steps, optionally surrounded by an if statement that checks a conjunction of a list of conditions (Cond). If any of the conditions are not satisfied, the loop does not perform any Steps on the data. If a loop Body contains an if statement in this manner, the loop is referred to as a conditional loop. Vars defined inside a Body are subject to the Scoping restriction defined in Section 3.1.3.
- **Cond**: A condition used in a conditional loop, which checks for equality between a LoopVar and a Var. The LoopVar and Var in a Cond are subject to the Conditional Loop restrictions defined in Section 3.1.3.

Having defined each of the nonterminal symbols, I define additional restrictions which must be respected by the variables used in the NERO domain-specific language.

### 3.1.3 Variable Restrictions

To allow NERO to infer the functionality of seed programs written in the NERO DSL, variables in the DSL are required to follow a set of restrictions, defined below.

- **Single Assignment**: No Var is allowed to be assigned to in more than one location. (A Var defined in a loop can be assigned to once per iteration, but otherwise Vars are only assigned to once.) Disallowing Var reassignment means that the value stored in each Var is retrieved by a specific Step, which greatly simplifies the inference algorithm. As an additional corollary, the Single Assignment restriction prevents a Var defined outside of a loop from being assigned to inside the loop, which is discussed further below.

- **Scoping**: No Var (including LoopVar) defined inside a loop can be accessed outside the loop.

Together, the Single Assignment and Scoping restrictions prevent Steps inside a loop from affecting Steps outside the loop, greatly simplifying the inference algorithm.

- **Conditional Loop**: This restriction is a set of three restrictions on the two variables compared in Conds.

  - The LoopVar compared in a Cond must be retrieved in the immediate surrounding loop, i.e., in the Iterate associated with the Body containing the Cond.

  - The Var compared in a Cond must be defined prior to the start of the immediate surrounding loop, i.e., prior to the Step containing the Body containing the Cond.
- The LoopVar and Var compared in a Cond must be of the same type.

The Conditional Loop restriction allows NERO to regenerate a conditional loop in the regenerated program as a SELECT query with a WHERE or JOIN clause. In both of these cases, a value retrieved from the table accessed by SELECT is compared to a value not from the table accessed by SELECT.

This concludes the definition of the grammar of the NERO domain-specific language. Next, I discuss the scope of programs allowed by this grammar.

### 3.2 Scope

Broadly speaking, the NERO domain-specific language supports programs which perform a sequence of operations to look up or store information. Seed programs are intended to mimic programs which access databases. In other words, the NERO domain-specific language supports programs which perform sequences of operations similar to database queries.

A strength of the NERO domain-specific language is that programs are allowed to perform long or complex sequences of lookup operations, where each operation depends on the result of one or more previous operations. Programs can look up or store information based on related information (for example, the student registration application presented Section 2.3.3 retrieves course information based on a student’s registration information). NERO can infer the functionality of such programs because the instrumentation (Chapter 5) allows the inference algorithm to observe each of the lookup operations performed. In contrast, a fully black-box algorithm would have substantial difficulty with programs which perform long sequences of operations.

The primary limitation of the NERO domain-specific language is in the conditionals which are supported. NERO is able to infer conditions, but only those which are conjunctions of equality checks. For example, the student registration application in Section 2.1.1 uses a conditional statement to perform a lookup operation. As a more complex example, two of the benchmark programs (Sections 8.1,8.2) authenticate
the user based on a password lookup before proceeding in each command. Nero is able to infer the functionality of the student registration application as well as both benchmarks. On the other hand, Nero would not be able to infer, for example, programs which query for dates prior to a given timestamp, or which use while loops to iterate until some condition holds. Although the instrumentation observes each data structure access performed, the Nero instrumentation (Chapter 5) does not directly capture conditional operations. Therefore, it is not straightforward to infer conditionals, and Nero uses a complex algorithm described in Section 6.6.1 for this purpose. Nero is only able to infer a restricted set of simple conditionals, where the algorithm is able to consider the entire set of allowed conditions and quickly narrow down the set of possibilities to infer the true condition.²

Despite this limitation, if Nero is used correctly, it is conceptually able to operate on programs with more complex logic than what is allowed by the domain-specific language. If a program is written in a modular fashion, where all database accesses can be isolated to a set of relatively straightforward lookup operations, it is possible to apply Nero to regenerate only the portion of the program which directly interacts with the database.

Strictly speaking, Nero as presented in this thesis is not well-suited for regenerating only a portion of a program, because both the seed program and the regenerated program output values by printing to stdout. However, it is straightforward to modify Nero to regenerate a program which returns its outputs instead of printing them. In this case, the seed program would still print output to stdout; only the regeneration algorithm needs to be modified. Every time the original regenerated program would print something, instead the modified regenerated program will append the value to an “output” list. The modified regenerated program will return the output list at the end of the execution. With this modification, the regenerated program can be used as a module in a more complex program.

²The importance of instrumentation to Nero is shown in this contrast: Nero supports programs which perform long sequences of operations because of instrumentation, but Nero has limited support for conditionals because of a lack of instrumentation.
3.3 Syntactic Flexibility

Because NERO does not read the source code, NERO supports seed programs that implement (in any way the user chooses) behavior expressible in NERO’s domain-specific language. More precisely, NERO supports seed programs for which there must exist a semantically equivalent program which:

- can be expressed in the NERO domain-specific language, and
- produces the same sequence of function calls to the global data structures and print operations as the seed program.

For example, instead of implementing a command with two nested loops, the user may implement this command with the outer loop where each iteration invokes a helper function that performs the inner loop. NERO infers the same results for both of these seed programs. Section 2.3.3 presents another example, where instead of executing a loop body when a condition holds, a `continue` statement executes to skip the loop body when the condition does not hold.
Chapter 4

Seed Program Interface

Because Nero uses a gray-box dynamic inference algorithm instead of reading the source code, it requires the user to use interface files to specify additional information about the seed program. There are two interface files:

- A *command specification* file (see Figure 2-2 in Chapter 2 for an example). This file describes the commands which the user wishes to regenerate. Nero requires this file because it does not analyze the seed program source code, so it does not know a priori what commands are implemented in it or what the types of the command arguments are.

- A *data specification* file (see Figure 2-3 in Chapter 2 for an example). This file describes the data structures used by the seed program and the desired database schema in the regenerated program. Because Nero does not analyze the source code, nor does it ask the user to provide sample data, Nero does not have information on the form of the data structures used by the seed program. Additionally, Nero uses this file to name database columns in the regenerated program, which otherwise would need to be generated in a systematic and potentially difficult-to-read manner.

Both of these files are straightforward for the user to write. They do not require the developer to understand or otherwise analyze the program themselves. (In contrast, for example, if Nero asked the user to provide test cases it would use for inference,
that would require the user to understand the seed program deeply enough to write good test cases.

I describe each of these files in more detail below.

4.1 Command Interface Generation

The user provides a command specification text file that describes the command interface of the seed program. NERO uses this specification file for two purposes:

- To determine what Python functions implement commands NERO should run inference on
- To generate a command loop which reads commands from stdin and dispatches them to the corresponding Python functions in the seed program. A command loop is also generated fulfilling the same purpose in the regenerated program. The command loop is described in more detail below.

Each line of the file specifies the interface for a single command, with the following information:

- The name of the command. NERO will, in both the seed program and regenerated program, generate a command loop which reads commands from stdin and dispatches them to the corresponding Python functions. The name of the command is what the user will type to indicate which command is run.

- The name of the Python function in the seed program implementing the command. The name of the function can be, but is not required to be, the same as the name of the command. Identifying the name of the Python function allows NERO to identify which function in the seed program implements each command, allowing NERO to invoke the correct function for each command without analyzing the seed program source code.

- The signature of the command, i.e. the list of types of the command parameters. Command parameters are allowed to be integers (“int”) or strings (“str”). Specify-
ing the signature of the command is needed because NERO does not analyze the seed program source code to determine the expected types of input parameters.\footnote{Even if it did analyze the source code, Python is dynamically typed so identifying the argument types would be difficult.} NERO uses this information to determine what type of arguments to generate when executing the seed program during the inference algorithm.

For ease of parsing, the prototype implementation of NERO expects the command specification file to contain a JSON list, where each list entry corresponds to a single command. Each command is described by a list, containing first the name of the command (as a string), then the name of the Python function (as a string), then the types of each command parameter (as a string, either “str” or “int”). It would be straightforward to modify the NERO implementation to accept a more human-readable or human-writable command specification file instead. Figure 2-2 presents an example of a command specification file.

Before starting inference, NERO preprocesses the command specification file and augments the seed program with an automatically generated command loop that parses input commands and parameters according to their interfaces. With this command loop, the augmented seed program works as follows. When the augmented seed program executes, it reads in a command (as specified by the name of the command), parses the input parameters, and enters the function that implements the command. The function writes the return value out to standard out. After inference is complete, NERO will generate a corresponding command loop in the regenerated program, which reads in commands and dispatches them to the corresponding Python functions in the regenerated program.

\section*{4.2 Data Structure Manipulation and Translation}

NERO uses a \textit{data specification} text file to determine the structure of data it should populate each data structure with, as well as to determine the database schema in the regenerated program. Additionally, NERO uses this file to generate a set of \textit{fill}
command handlers, which are included in the command loop generated by NERO. Fill commands allow NERO to populate data structures with arbitrary contents during the inference process. More information on fill commands is given towards the end of this section.

The data specification file specifies the data structures used by the seed program and the form of these data structures, as well as the database schema in the regenerated program. Each data structure in the seed program mimics a database table in a manner described below. In the regenerated program, NERO will regenerate a database table for each data structure in the seed program. For each data structure and corresponding database table, the data specification file specifies:

- The name of the data structure (more specifically, the name of the global variable storing the data structure in the seed program).
- Whether the data structure is a list or a dict
- The name of the corresponding database table in the regenerated program. The name of the table is allowed to be, but is not required to be, the same as the name of the data structure.
- The name and type of each column in the corresponding database table.

The data structures mimic database tables, as follows: Each data structure is either a list of tuples or a dict where the values are tuples. A tuple corresponds to a row in the corresponding database table, so the tuples within the same data structure are required to have the same length and type signature. Each position in the tuple corresponds to a column of the database table, and has the same type as its corresponding column. The database table also contains an additional column: a primary key. If the data structure is a list, the primary key column is of integer type and corresponds to the list index; if the data structure is instead a dict, the primary key column is of string type and corresponds to the dict key, which is of string type.\(^2\)

\(^2\)Because of a quirk in JSON, which is used by the fill interface to specify data structure contents, NERO requires dict keys, and therefore primary keys of tables corresponding to dicts, to be of string type. This restriction is straightforward to remove, however.
(It follows that the number of columns in each database table is one more than the length of the tuples stored in the corresponding data structure.)

Because of the close relationship between the form of each data structure and the form of the corresponding database table, specifying the type of each column in the database table implies the form of the data structure, including the types of the values stored in the data structure. Therefore, the data specification file does not need to explicitly indicate the form of each data structure.

For ease of parsing, the prototype implementation of NERO expects the data specification file to be in a different format from the one described in this section, although the information contained within is equivalent. The prototype NERO implementation expects the data specification file to contain a JSON object, where each key-value pair corresponds to a single data structure-database table pair. Each key is the name of the data structure, and the corresponding value is a list containing three entries: a “template,” a list of database column names, and the name of the corresponding database table. The template is defined as follows:

- If the data structure is a list, the template is a JSON list containing a single item. This item is a JSON list representing a tuple, as described below.

- If the data structure is instead a dict, the template is a JSON object containing a single key-value pair with key “str” and value a JSON list representing a tuple, as described below.

The “tuple” in the template contains entries which are either “str” or “int” depending on the type of the corresponding tuple entries in the seed program data structure, or equivalently, the type of the corresponding column in the regenerated program database table. The length of the “tuple” in the template is the length of the tuples in the corresponding data structure, and one less than the number of columns in the corresponding database table (as described above, the database table has an additional column corresponding to a list index or dict key). See Figure 2-3 for an example of a data specification file.

\[^3\]Again, to accommodate a quirk of JSON, NERO requires all dict keys to be strings and not integers
As with the command specification file, it would be straightforward to modify the NERO implementation to accept a more human-readable or human-writable data specification file instead.

Before starting inference, NERO preprocesses the data specification file and augments the seed program with automatically-generated *fill command* handlers to help to populate values into the list/dict data structures. The fill command handlers are integrated into the command loop (Section 4.1). One fill command handler is generated for each data structure. Each fill command handler takes in a file name as a parameter. It locates the data structure by global variable name, then replaces the contents of the data structure with the contents of the specified file (as given in JSON format). By allowing NERO to populate data structures with arbitrary values, the fill interface allows the inference algorithm to fully explore code paths and infer the seed program’s function unambiguously.

---

4Because JSON expects object keys to be strings, this interface only works when dicts have string keys, which is why NERO requires dicts to have string keys.
Chapter 5

Data Structure Instrumentation

To observe the dynamic data access behavior of the seed program, Nero instruments the seed program so that each execution of the seed program produces a *trace*. Following the domain-specific language presented in Section 3.1, Nero models the seed program’s functionality as a sequence of function call events, where function calls can be either data structure accesses (e.g., list index operations), print statements, or loops over data structures. The trace, described in the sections below, is a concrete instantiation of the sequence of function call events. At a high level, to perform inference, Nero runs the seed program several times to observe several traces, and infers an abstract sequence of function call events which fits all of the traces. The abstract sequence of function call events is the inference result.

In Section 5.1, I describe the structure of the trace and how function calls in general are recorded in the trace. In Section 5.2, I describe how loops in particular are recorded in the trace. In Section 5.3, I describe how crashes due to KeyErrors and IndexErrors are recorded in the trace. In Section 5.4, I describe how the instrumentation is implemented.

5.1 Recording Function Calls

The Nero instrumentation produces an execution log, or *trace* for all data structure accesses (including loops over data structures) and calls to the print function. Col-
lectively, I refer to data structure accesses and calls to the print function as function calls. I describe how the trace is structured below.

In Python, every list and dict data structure access is performed by calling a function, e.g., a list index operation is `__getitem__()`. I refer to the functions which are called in this way as accessor methods.

The instrumentation records information on each data structure access in a separate line of the trace. For each data structure access, the instrumentation records:

- The type of the data structure (i.e., list or dict)
- The UID (see below) of the data structure
- The accessor method being called
- The concrete values of the arguments passed to the accessor method
- The concrete value returned by the accessor method, if any
- If the accessor method is `__next__()`, which marks the start of a new iteration in a loop, additional iteration information is recorded (See Section 5.2).

For each call to the print function, information is recorded in the same format as data structure accesses. In this case, the type of data structure, UID, and accessor method are all recorded as “print.”

The instrumentation logs this information to a trace file read by the inference algorithm, with one line corresponding to each function call. See Section 2.3.2 or Section 2.3.3 for an example.

The instrumentation does not know the name of each data structure, so it uses unique identifiers (UIDs) to indicate which data structure is accessed in each line of the trace. The instrumentation assigns a UID to every data structure instance that appears during execution. UIDs are assigned in the order that each data structure is initialized, i.e., UID 1 is the first data structure initialized in the seed program. Recall that the seed program initializes global data structures at the beginning of the execution and that these initializations happen in a deterministic order (Section 3.1).
Consequently, the NERO instrumentation will assign small and deterministic UIDs to all global data structures in the seed program which are preserved if the seed program crashes (e.g., due to a KeyError or IndexError) and is relaunched. (See Section 5.3 for more information on crashes.) The NERO inference algorithm, in a preprocessing step at the start of inference, determines which UIDs refer to which data structures (See Section 6.1 for more information.)

NERO uses the information in this trace to perform inference. To perform inference, NERO must determine what sequence of function calls will be made by the seed program, and how the seed program constructs arguments to these function calls. Each piece of information in the trace contributes towards this objective:

- The data structure type and UID as well as the name of the accessor method directly indicate to the inference algorithm what sequence of function calls is being made.

- The concrete values passed into and returned by each function call allow the inference algorithm to narrow down and identify how the seed program constructs arguments to each function call. See Section 6.4 in the Algorithm chapter for more details, or Subsection 2.3.2 in the Example chapter for an example.

The information recorded in the trace, as presented in this section, allows NERO to infer flat sequences of data structure accesses and print statements. Next, I present what information is recorded in the trace to allow NERO to infer the functionality of seed programs which contain loops.

### 5.2 Recording Loops

NERO’s inference algorithm supports seed programs which iterate over data structures (Section 3.1). When the seed program contains a loop, NERO needs to infer:

- What data structure is being iterated over.

- What sequence of function calls occur in the body of the loop.
• How the arguments to these function calls are constructed. (More specifically, NERO needs to know what value is retrieved from the data structure on each iteration. This information is sufficient to perform the inference procedure mentioned in Section 5.1 and described in Section 6.4.)

The instrumentation records information in the trace to allow NERO to perform the necessary inference. In Python, when a program iterates over a data structure (using a `for` loop), the `__iter__()` function is called on the data structure which returns an `iterator` object. At the beginning of each iteration, the iterator’s `__next__()` function is called, which returns the next item from the data structure. Neither function has any input arguments. Function calls to `__iter__()` are recorded in the trace in the same manner as other data structure accesses discussed in Section 5.1. Function calls to `__next__()` are also recorded in a similar manner; the differences are discussed below. The record of `__iter__()` and `__next__()` function calls indicates to the inference algorithm what data structure is iterated over at each step in the seed program. A function call to `__next__()` marks the beginning of each iteration, allowing the inference algorithm to determine what sequence of function calls occur in the body of each loop. The return value of `__next__()` allows the inference algorithm to know which element is being iterated over, and thus what retrieved value the loop body has access to. See Section 6.5 for more details on how inference is performed on the loop body, or Section 2.3.3 for an example.

Function calls to `__next__()` are recorded in a manner slightly different from other function calls. The function `__next__()` is a method of the iterator returned by `__iter__()`, not a method of the list or dict itself. The owner of the method does not make a significant difference; the trace records the UID of the list or dict which generated the iterator. However, the trace must record additional information about `__next__()` function calls for the inference algorithm to correctly infer what sequence of function calls occur in the body of each loop. In particular, consider a situation where a seed program contains two nested loops over the same data structure. Then when `__next__()` is called, the instrumentation will record the data structure being looped over. This information is not sufficient to determine whether the inner or outer
loop is advancing, because the same data structure is being looped over in both cases.

To unambiguously distinguish between nested loops over the same data structure, the NERO instrumentation records a iteration ID for each call to \_\_next\_\_(\). When the \_\_iter\_\_() function returns an iterator, the instrumentation assigns the iterator an ID. An ID of 0 is assigned to the iterator returned from the first \_\_iter\_\_() call to a particular data structure (i.e., the first loop over that data structure), and the ID increments by one for each call thereafter.\textsuperscript{1} When the \_\_next\_\_() function is called on an iterator,\textsuperscript{2} the instrumentation records the ID of that iterator, i.e., the iteration ID. The iteration ID allows the NERO inference algorithm to distinguish between inner and outer nested loops over the same data structure, because their iteration IDs are different. In the trace, for function call other than \_\_next\_\_(\), the instrumentation records a $-1$ as a placeholder for the iteration ID.

One more piece of information is captured on each call to \_\_next\_\_() to support the use of the enumerate function. As described in Section 3.1, a seed program may call the enumerate function when iterating through a list. The enumerate function is a native Python function which wraps a list being iterated over. Where iterating over a list typically only returns each item stored, enumerate will additionally return the list index of the item. Because enumerate is a built-in Python function and not a list accessor method, the instrumentation does not record it. However, enumerate does call the \_\_next\_\_ function, which is recorded by the instrumentation. The NERO inference algorithm assumes that each loop over a list may be called with enumerate, so it assumes that the seed program has access to the list index. Therefore, in each call to \_\_next\_\_(), the instrumentation records the index of the iteration, which is 0 the first time \_\_next\_\_() is called on each iterator, and increments by one each time

\textsuperscript{1}This assignment algorithm might cause two iterators from two different data structures to have the same ID. Also, IDs are reset to 0 if the seed program crashes and is restarted. This behavior is not a problem, because the iteration ID is only needed to distinguish between nested loops over the same data structure in a single execution of the seed program.

\textsuperscript{2}In particular, the iteration ID is not recorded on a call to \_\_iter\_\_(). There is no particular reason it is not recorded; however, recording this information is unnecessary because each call to \_\_iter\_\_() is immediately followed by a call to \_\_next\_\_. For this reason, the \_\_iter\_\_() function is itself unnecessary to record in the trace, but it simplifies the instrumentation and inference slightly to record it anyway.
thereafter. When iterating over a list, the index of the iteration recorded equals the list index returned by `enumerate`. In the trace, for function call other than `__next__()`, the instrumentation records a $-1$ as a placeholder for the iteration index.

Recall that seed programs are allowed to contain conditional loops (Section 3.1), where the body of the loop only executes if a certain condition is met. Notably, the instrumentation does not directly capture any information on the condition. Instead, seed programs are required to have non-empty sequences of function calls in the body of each loop. Therefore, if the condition is not met, the instrumentation will record an empty loop body (i.e., after the call to `__next__()` marking the start of the loop, there is nothing in the trace before the next call to `__next__()`). This information is sufficient to allow Nero to detect the presence of a conditional loop as well as whether the condition is satisfied or not in each execution. Through repeated execution, Nero is able to cause the condition to satisfy and infer the contents of the loop body. By observing whether the condition is satisfied or not when the seed program is executed with different inputs and data structure contents, Nero is able to infer the condition of the loop. See Section 6.6 for more details on how Nero infers the loop body and the condition, or Section 2.3.3 for an example.

### 5.3 Recording Crashes

Because seed programs contain list index and dict key lookup/store operations, they will sometimes crash due to an IndexError or KeyError. (Seed programs allowed by Nero (Section 3.1) do not contain any other constructs which may lead to a crash.) In this case, the Nero instrumentation records the following information:

- The type of the data structure (list or dict)
- The UID (see Section 5.1) of the data structure
- The name of the accessor method which raised the exception

---

3When iterating over a dict, the index of the iteration is unused.
• The values of the arguments passed to the accessor method

The type of the data structure indicates whether an IndexError or KeyError is raised. (The name of the accessor method, therefore, does not actually need to be captured, but it is convenient to record it anyway, especially if NERO is expanded to support additional constructions in the future which may crash in other ways.)

The information about the crash is recorded as the last line of the trace. Thus, the trace records all function calls which occurred before the crash, as well as information about the crash itself.

With the information recorded in the trace, the inference algorithm is able to determine the cause of the crash and is able to impose a constraint to prevent the crash from happening in the future; for example, if an IndexError occurs when a command parameter is used to index into a list, the inference algorithm will detect this and ensure that in future executions, the command parameter will be less than the length of the list. Preventing crashes is essential for the seed program to run to completion, so that NERO can infer its full functionality. See Subsection 2.3.2 in the Example chapter for an example, or Section 6.3 for a more detailed explanation.

The information recorded by the instrumentation on function calls, loops, and crashes provides the NERO inference algorithm with enough information to infer the functionality of the seed program. I conclude this chapter with a note on how the NERO instrumentation is implemented.

5.4 Implementation

This instrumentation is implemented as a set of wrappers,4 in which list and dict data structures (as well as the print function) are replaced by instrumented versions which wrap the original accessor methods and log all accesses to a file. NERO applies this instrumentation to the seed program by making use of Python’s built-in exec

---

4As an aside, when NERO was originally under development, I tried instrumenting the C Python implementation directly. Instrumenting the C Python implementation worked out very poorly, because the C Python implementation heavily uses dicts for internal storage, which leads to a large amount of noise in the log, and in a few cases broke the interpreter entirely.
function, which allows the interpretation of arbitrary Python code. Namely, NERO first runs the code which defines these wrapper classes which override list, dict, and print, then calls `exec` on the seed program. (As described in Chapter 4, NERO also augments the seed program with some additional functionality, namely a command loop and fill command handlers. This augmentation process is done by concatenating the code implementing the additional functionality to the seed program before calling `exec`.) To make the instrumentation work fully, NERO first makes use of the `ast` package to preprocess the seed program and ensure that all lists and dicts accessible to the seed program are replaced by the instrumented versions.
Chapter 6

Inference Algorithm

After augmenting the seed program with a command loop (Section 4.1) and fill command handlers (Section 4.2), and instrumenting the seed program (Chapter 5), Nero runs an inference algorithm to infer the functionality of the seed program. As a preprocessing step, the inference algorithm learns the UID, assigned by the instrumentation, for each data structure (Section 6.1 below). Then, the algorithm performs inference on each command individually.

Nero is able to infer commands individually because the only state which persists between multiple runs of a command or between different commands is the global data structures. (Even if, for example, two commands share a single helper function, Nero is oblivious to this fact, and this sharing does not prevent Nero from inferring the commands independently because it is only a sharing of functionality, not a sharing of state.) The fill command handlers generated by Nero populate the data structures with data specified by the inference algorithm. Nero uses the fill command handlers to completely repopulate the global data structures with fresh data before each execution of any command in the seed program. Therefore, each command execution does not affect any other command execution. Because executions do not affect each other, Nero can treat each command in isolation, greatly simplifying the inference process.

To infer each command, the Nero inference algorithm executes the seed program multiple times using different input parameters and data structure contents. Figure 6-1 presents how Nero infers each command at a high level. Each time the seed program
Steps to infer a command of the seed program:

1. Populate global data structures and prepare input parameters, using values of the appropriate data types.

2. Analyze the trace. If the program crashes due to a KeyError or IndexError, learn which operation caused the crash. Populate data structures again so that the same error should not happen again (Section 6.3). Run the command again, until the program finishes an execution without crashing. Note that there could still be a potential crash that only occurs conditionally, and this condition may not have been inferred yet.

3. Analyze the trace to infer the following properties of the command:
   
   (a) The sequence of function calls made to the global data structures.
   
   (b) For each function call made to a data structure, determine where the function parameters came from (Section 6.4). The data source can be an input parameter or a value returned from a previous function call.\(^1\)
   
   (c) Presence of unconditional loops (Section 6.5).
   
   (d) Presence of conditional loops (Section 6.6).

4. Repeatedly run the command with carefully chosen values for data structures and input parameters (Section 6.2). Progressively infer more functionality of the seed program. Stop when confident about the inference outputs.

Return the inferred command.

Figure 6-1: Inference algorithm outline
is executed, the instrumentation records the series of resulting data structure accesses in a trace (Section 5.1), indicating which data structure was accessed, which accessor method (i.e., function) was called to access the data structure, with what concrete arguments it was called with, and what values were returned, if any. NERO analyzes the trace to determine the next set of command parameters and data to use to gain more information on the seed program, and ultimately to infer the functionality of the seed program.

6.1 Learning UIDs of Data Structures

Before performing inference on any of the commands, NERO must learn the correspondence between data structure names and UIDs. NERO’s instrumentation generates a UID for each data structure (Section 5.1), but does not know the name of each data structure because it does not analyze the source code to identify the variable a data structure is stored in. Conversely, the interface specification files written by the user (Chapter 4) indicate the form of each data structure and the global variable each data structure is stored in, but does not know the corresponding UIDs.

Learning the name-UID correspondence is necessary for two reasons:

- **NERO** intelligently populates data structures with data throughout the inference process using fill commands which correspond to data structures by name, but the instrumentation only refers to data structures by UID. For example, the inference algorithm might learn, after performing a small amount of inference on the trace reported by the instrumentation, that values from the first column of a list with UID 2 are used to index into a list with UID 3. Therefore the inference algorithm, to avoid an IndexError, should populate the first column of UID 2 with small values. But to do this, NERO needs to know the name of the data structure with UID 2, so the appropriate fill command can be used.

- The regenerated code will refer to database tables by name, but the instrumentation refers to data structures by UID. For example, the instrumentation might
indicate that the first step in a command is to perform a list index lookup on a
list with UID 1. To regenerate a corresponding SELECT query, NERO needs to
know the name of the database table corresponding to the list with UID 1. The
data specification file only indicates the correspondences between database table
names and data structure names, so NERO needs to know the name of the data
structure with UID 1 before it can look up the corresponding database table.

To learn the UID of each data structure, NERO uses the fill command handlers
generated during the preprocessing step, which populate data structures referred to
by name with specified data. NERO runs the fill command for each data structure
in turn with random data; for each fill command, NERO knows the name of the
data structure which is published. The instrumentation will capture the population
operation and report that data has been inserted into a certain data structure, referred
to by UID. Matching the name of the data structure populated by the fill command
with the UID reported by the instrumentation gives the correspondence between
names and UIDs. Because data structures are initialized at the beginning of the
program in deterministic order, and UIDs are assigned by order of initialization, these
name-UID correspondences will remain stable across multiple executions even if the
seed program crashes and is restarted. Therefore, NERO only needs to learn name-UID
correspondences once for the whole seed program, at the beginning of inference.

I next present details of how the NERO inference algorithm analyzes the trace and
assigns new values to input parameters and data structures to infer each command
individually.

### 6.2 Choosing Values for Inputs and Data Structures

During the inference process, NERO will populate data structures with fresh values
prior to each execution of each command in the seed program. NERO contains a
randomized solver that chooses random values of the correct type (Section 4.2) to
assign to data structures of the seed program, subject to the following constraints:
• Values used to index into a list must be within the range of the list indices. Values used as a key in a dict must exist in the dict (Section 6.3).

• Certain values must equal, so that potential conditional loops may execute.

These constraints are unknown initially and generated during the inference algorithm. Subject to these constraints, the solver generates values which vary as widely as possible in each execution, to resolve ambiguity (Section 6.4.1) and to discover the conditions for conditional loops (Sections 6.6 and 6.6.1).

The general form for the constraints used in the inference algorithm is:

“Column A of data structure X should be a subset of column B of data structure Y.”

These constraints are never imposed between two columns of different types.

Note that this category of constraints is always satisfiable, even if they are imposed in a circular fashion, because one could populate all data structures with the same set of elements (for a given type), in which case each column is a subset of every other column (of the same type). A slightly better algorithm is the following:

1. Choose a fixed length $l$ to use for all of the data structures.

2. Determine the undirected constraint graph, where vertices are data structure columns and edges are subset constraints between columns.

3. For each connected component, generate a set of random values depending on the length of the data structure. (If a column in this connected component is a column of list indices, the range $[0, l - 1]$ is used instead.)

4. For each column (vertex) in each connected component, populate it with a permutation of the set of random values.

---

2The prototype implementation of NERO does not actually use this algorithm, for reasons related to NERO’s development history. NERO’s implemented algorithm is more complex and difficult to analyze without leading to any substantive benefits, though in many cases it converges to this simpler algorithm. For this reason, I do not discuss the implemented algorithm in this thesis.
This solver algorithm can lead to some ambiguity in the seed program execution because many columns will contain the same set of values. However, in practice the ambiguity is not problematic when data structures are reasonably large (e.g., at least 6 elements). Furthermore, this ambiguity is unlikely to persist across several executions, which NERO will perform to eliminate ambiguity (Section 6.6.2).

During the inference algorithm, as NERO learns more about the seed program, the inference algorithm will adjust the constraints used by the solver, then repopulate data structures using the solver and execute the seed program. Adjusting constraints and repopulating data structures allows NERO to explore new code paths and reduce ambiguity in the inferred functionality of the seed program.

I next present several important steps of the NERO inference algorithm.

### 6.3 Constraints to Avoid Crashes

The NERO inference algorithm is designed to work with programs which implement only the core functionality desired by the user, the inference algorithm is designed to be able to infer the behavior of seed programs which do not perform any input validation to check for corner cases which may crash the seed program. For example, seed programs may crash due to an IndexError or KeyError when performing a list index or dict key lookup. As described in this section, NERO automatically handles the error and prevents it from occurring in future executions of the seed program. It is well-known that writing input validation code by hand is time-intensive and prone to error. Because the inference algorithm is able to infer the functionality implemented by seed programs which do not contain input validation code, the user is able to focus on implementing the core functionality they desire and is relieved of the burden of writing input validation or error-checking code.

As NERO populates different values for the seed program data structures and runs the seed program, the program may crash from a KeyError (from a dict lookup) or IndexError (from a list lookup). A crash prevents the full execution of the seed program, so NERO needs to impose constraints on the seed program (of the form
described in Section 6.2) to prevent the crash from occurring in the future. KeyErrors and IndexErrors are the only possible errors which may occur for seed programs conforming to the Nero domain-specific language (Chapter 3). If an error other than an IndexError or KeyError occurs in the seed program, Nero will detect the unknown error, halt the inference algorithm, and report an error message.

When a KeyError or IndexError occurs, the Nero instrumentation (Section 5.3) detects the error and records information about the operation that caused the error. This information is logged in a trace, which records all data structure accesses which occurred prior to the error (i.e., what structures were accessed, the accessor methods used (i.e., what functions were called to access data structures), the arguments given to the methods, and any return values), as well as information on the error itself (i.e., the structure which was accessed at the time of the crash, the accessor method which caused the crash, and the arguments passed to the accessor method which caused the crash).

After an error occurs, the inference algorithm identifies the source of the input arguments passed to the crashing function call by comparing the value of the input arguments to the command parameters and values retrieved from prior data structure accesses. The inference algorithm then imposes a constraint to prevent this error from happening in future executions of the seed program:

- If the error was an IndexError, the new constraint forces the source of the argument to be less than the length of the list being accessed. If instead the error was a KeyError, the constraint forces the source of the argument to be one of the keys present in the dict being accessed.

- If the source of the argument which caused an error was a command parameter, the new constraint is imposed on the command parameter. If instead the source of the argument was a data structure access, i.e., the argument was a value retrieved from a certain column of a data structure, the constraint is imposed on that column of the data structure. (In this case, the values in the column of the source data structure are constrained to be a subset of the list index column
See Section 2.3.2 for an example of the inference algorithm imposing a constraint in response to an IndexError.

In some cases, the source of the argument may be ambiguous, for example if the value passed as an argument occurs multiple times as command parameters or values returned from prior data structure accesses. In this case, the inference algorithm tries imposing a constraint on each candidate argument source, one at a time, until the error stops occurring.

These constraints prevent the error from occurring in the future because the domain-specific language used by Nero (Section 3.1) heavily restricts the use of conditionals in the seed program. In particular, the source of an argument (whether a column of a data structure or a command parameter) is always the same between different executions of the same command. Therefore, constraining the source to be a valid index or valid key prevents the error in all future executions. As noted above in Section 6.2, these constraints are always satisfiable.

Once crashes have been prevented and the command runs to completion, the inference algorithm proceeds to identify the sources of each argument passed to function calls (data structure accesses and print statements) in the seed program. If a crash occurs in the future, for example due to an IndexError in a conditional loop which has so far not executed, the inference algorithm will go back to this step and resolve the error before continuing.

### 6.4 Identifying Data Sources

The purpose of the inference algorithm is to determine the program’s functionality in terms of a sequence of function calls (data structure accesses, print statements, and loops), and the sources for each of the inputs to these function calls (e.g. “the input to this function call is the first command parameter”). To do so, Nero executes the seed program command and analyzes the resulting trace of function calls recorded by the instrumentation. This trace (as given in Subsection 2.3.2 as an example and defined
in Chapter 5) indicates the sequence of function calls made during the execution. For each data structure access, the trace indicates which data structure is accessed, the accessor method (i.e., function) called, the concrete inputs to accessor methods, and any concrete return values. Similar data is recorded for each print statement or loop.

NERO abstracts away each concrete value in the trace and replaces it with the data source that the value came from. Specifically, NERO keeps track of all the values available to the seed program as a context. Every time a function call completes, NERO adds the return values to the context. The context also contains all input parameters for the command.

To infer the data source for any concrete value in the trace, NERO searches through the context to identify where the value was made available (the context location of the value). If there is only one such location, NERO has found the data source for this concrete value.

### 6.4.1 Resolving ambiguity

Sometimes, there may be ambiguity when searching for the data sources. Ambiguity occurs if the same concrete value occurs multiple times in the context. To resolve ambiguity, NERO uses the solver to assign input parameters and data structure contents using random values that vary widely, subject to necessary constraints (summarized in Section 6.2). Each execution of the seed program is likely to be assigned a different set of random values. After several such executions, the false ambiguities are unlikely to persist. When ambiguity exists, the inference algorithm repeatedly executes the seed program until the ambiguity is resolved. If the ambiguity is still not resolved after several executions, the inference algorithm concludes that the ambiguity is a true ambiguity inherent in the seed program (e.g., as might occur if the same data structure is accessed twice with the same index) with high probability. (See Section 6.6.2 for more details.) Consequently, the inference algorithm reaches a confident inference about the data sources for all concrete values that appeared in the trace.

Algorithmic simplicity and flexibility are the primary reasons NERO uses a randomized approach to generate data and resolve ambiguity. Instead of a randomized
approach, there are two candidate deterministic approaches I considered, both of which had disadvantages compared to the simple and easy-to-implement randomized approach.

- One approach is to reason about the flow of values in the seed program execution, and where ambiguity exists, solve for a modification to the data to resolve the ambiguity. However, this would require heavyweight analysis capable of tracing the path of values through a potentially complex seed program. The analysis would also need to be able to detect and ignore true ambiguity arising in the seed program (for example, if the seed program accesses the same data structure twice with the same index or key). The amount of analysis required would add substantial complexity to the algorithm.

- Another approach is to deterministically generate a set of unique values for each execution. However, compared to the fully randomized approach, this method is a little more difficult to implement, does not offer considerable advantages, and has a few disadvantages. In the randomized approach, generating values in a large range will cause each individual execution of the seed program to only access a sparse set of the possible random values. The sparseness of the values accessed makes the probability of a collision vanishingly small, especially persistent collisions over multiple executions. In contrast, the deterministic approach, if not done carefully, could result in systematic collisions. One example is if numerical values were naively generated starting at 0 and incrementing by 1 for each value, there would be conflicts between generated values and list indices. More importantly, the randomized approach extends well to a temperature-based technique for inferring unknown conditions in the seed program, as elaborated on in Section 6.6.1. In contrast, extending the deterministic approach to infer unknown conditions would require substantial complexity.

\[\text{For simplicity, prototype NERO implementation generates values in a large but fixed range of values, so a sufficiently complex seed program will cause collisions to occur. In future work, it is straightforward to scale the range of possible random values with the complexity of the inferred seed program.}\]
6.5 Identifying Loops

NERO is able to infer the presence and contents of loops over data structures in the seed program. To loop through elements in a data structure, the Python interpreter invokes the data structure's function `__iter__()` at the beginning of the loop to retrieve an iterator and invokes the iterator's function `__next__()` at the beginning of each individual iteration. Both of these function calls are captured by the NERO instrumentation. This information allows the inference algorithm to identify loops in the trace.

The instrumentation records each call to `__iter__`, including which data structure is being iterated over, allowing the inference algorithm to learn that a new loop is beginning. `__iter__` has no input arguments, and its return value is not meaningful.

The instrumentation records information on each call to `__next__`, allowing the inference algorithm to determine which loop has advanced an iteration. To determine which loop has advanced an iteration, the instrumentation records which data structure is associated with the `__next__` call. However, it is possible there are two nested loops in the seed program over the same data structure. Therefore, the instrumentation captures additional information: the iteration ID. This ID is, within a single command, unique for each individual loop, so in the nested loop scenario, the iteration ID allows the inference algorithm to distinguish between the inner and outer loop's `__next__` calls. See Section 5.2 for more details on how the iteration ID is captured.

The instrumentation also records additional information on each call to `__next__` to allow the inference algorithm to determine what values in the data structure are being accessed. The return value of each `__next__` call is the item in the data structure retrieved for that iteration. However, seed programs (specified in Chapter 3) are also allowed to use the `enumerate` function when iterating over lists, which returns the list index corresponding to each item in the list as well as the list item itself. Because the `enumerate` function is not a list method, the instrumentation does not capture it directly, although `enumerate` calls the `__next__` function, and these calls are still recorded. Instead, for each call to `__next__`, the instrumentation captures the
index of the iteration (corresponding to the list index if the data structure is a list). See Section 5.2 for more information on how the index is captured. The inference algorithm always assumes, when iterating over a list, that the seed program may be using `enumerate` and therefore has access to the list index. Finally, the inference algorithm adds the retrieved data structure item, and list index if appropriate, to the context, which records all values accessible to the seed program at each point in the execution.

The inference algorithm then infers the contents of the loop (the loop body) as follows. It starts by identifying the sequence of function calls for the first non-empty iteration\(^4\) in the trace. The trace for each iteration consists of all function calls that appear between two adjacent calls of the loop’s `__next__()` function.\(^5\) NERO recursively infers the functionality of this loop iteration. In the rest of this description, I refer to the inferred functionality as the inferred loop body.

After this initial inference result for one loop iteration, NERO uses the inferred loop body to “consume” the trace containing the remaining iterations of the loop, in the following manner. Because the seed program executes the same sequence of function calls in each iteration, each function call in the inferred loop body corresponds to a single line in the trace for each iteration. For each iteration, NERO performs inference on each line of the trace. Each inference result is then merged with the corresponding function call in the original inferred loop body to eliminate ambiguity.

### 6.6 Discovering Conditional Blocks in Loops

Seed programs (defined in Chapter 3) are allowed to contain conditional loops, where loops may contain a conditional block with certain conditions that correspond to SQL queries that use SELECT, WHERE, and JOIN operations. In these conditional loops, if the condition is not satisfied, the loop body will not execute. Therefore, to fully infer

\(^4\)Note that empty iterations are possible only for loops whose loop body is a conditional block. See Section 6.6.

\(^5\)If the first non-empty iteration is the last iteration, the end of the iteration block cannot be inferred, so NERO reruns the command with new inputs to try again.
the functionality of the seed program, NERO must cause the condition to be satisfied, so the loop body can be inferred. NERO must also infer the condition itself, so it can be used in the regenerated program. I present a randomized algorithm to infer the condition. Causing the condition to be satisfied is discussed below in Section 6.6.1.

The conditions which are supported by the inference algorithm are of the form:

“Column A of data structure X equals value B;”

where B refers to a value in the context (see Section 6.4 for definition), and X is the data structure being iterated over by the loop. Note that based on the subset of Python supported by NERO, value B is either a command parameter or a value returned from a data structure access, in which case it is a value stored in some column of some data structure. For example, a condition might be that “The value in the second column of a data structure matches the first input parameter passed to the command.” Each conditional block may contain conjunctions of these conditions. Note that, for any given conditional loop in the seed program, there are a finite number of possible conditions, because the data structure has a finite number of columns and there are a finite number of values available in the context.

The execution trace of a conditional loop may contain a sequence of iterations where some iterations are non-empty (where the condition is satisfied) and other iterations are empty (where the condition is not satisfied).\(^6\)

When inferring a conditional loop, NERO needs to infer the condition for entering the conditional block. NERO starts with all potential conditions and gradually rejects incorrect conditions after multiple executions. For each execution of the seed program command, the algorithm assigns carefully chosen values to input parameters and data structure contents. These random values are chosen so that it is likely for potential conditions to satisfy, so that the seed program may execute the body of the conditional loop (see Section 6.6.1). The algorithm then observes the trace of the seed program

\(^6\)For most iterations, if the iteration is empty, this is apparent in the trace because the next iteration begins immediately. However, it is not immediately clear from the trace whether the last iteration is empty. NERO considers both the possibility that it executed and that it did not execute. One of these possibilities will lead to an inference result which is inconsistent with inference results from other executions of the seed program; the other possibility will be consistent with inference results from other executions. NERO discards the inconsistent possibility.
and rejects the potential conditions that are inconsistent with the trace. Specifically, if an iteration enters a conditional block, the algorithm collects the conditions that were not satisfied for the iteration and rejects them. On the other hand, if an iteration does not enter a conditional block, the algorithm collects the conditions that were satisfied for the iteration and rejects them. The algorithm will repeatedly execute the seed program until only the true condition remains. See Section 6.6.2 for further details on how inputs to these repeated executions are chosen, and how many times the seed program will be executed.

6.6.1 Allowing Unknown Conditions to Satisfy

I have presented how to infer the conditions from a high level. Note, however, that if the inference algorithm were to naively choose arbitrary random values from a wide range, the unknown condition, which checks for equality of values, would be unlikely to satisfy and the inference algorithm would not be able to observe an execution trace of the loop body. This lack of observation would prevent the inference algorithm from discovering the body of the conditional loop. Consequently, the inference algorithm must carefully choose the random values.

The NERO inference algorithm first performs a quick check over possible conditions that are equality checks without conjunctions. For each such possible condition, the algorithm executes the seed program command using a value assignment that enforces constraints for this possible condition. The constraint is defined as follows:

Let $X$ be the data structure being iterated over. Let the hypothetical condition be “The value retrieved from column $A$ of $X$ equals value $V$,“ and let $V$ be retrieved from column $B$ of some data structure $Y$. Then NERO constrains column $B$ to be a subset of column $A$. If instead $V$ is a command parameter, then NERO constrains the command parameter to be an element in column $A$.

The constraint will force the hypothetical condition to satisfy at least once, because $V$ will necessarily be contained in column $A$, and the loop is iterating over all values in
column A. Therefore, if the condition under consideration were the correct condition, the seed program would successfully enter the conditional block during at least one iteration of the loop. NERO imposes each possible constraint in turn, until a correct condition is found. The constraint is kept for the future, so that the condition will always execute at least once in each future execution of the seed program. See Section 2.3.3 for an example of this constraint-based technique in use.

Unfortunately, this constraint-based methodology only works on simple equality checks and not conjunctions of equality checks. Generalizing the constraint-based methodology to support conjunctions would require the constraint solver (Section 6.2) to support more complex constraints than the subset-based constraints currently supported. It is also not clear that the more complex constraints required are necessarily always solvable. Instead, NERO falls back to a temperature-based algorithm that allows the unknown condition to be likely to satisfy.

This fallback algorithm uses a variable that I call the *temperature* to control the level of randomness in the assigned values for executing the seed program. A high temperature causes the solver to use a large range to assign random values, and additionally increases the size of data structures. A low temperature causes the solver to use a small range to assign random values. When the temperature is zero, all the assigned values are equal wherever possible.

Even at low temperatures, constraints on the data must be observed. These constraints include:

- Each data structure must contain at least 2 entries. This requirement is because, when the inference algorithm first infers the contents of a loop body, the inference algorithm must know where the loop body ends, which is only determined by the start of the next iteration, so this is not possible with a single iteration.

- List indices must be $[0, n - 1]$ where $n$ is the length of the data structure. Dict keys must be unique.

- Values must be of the correct type.
• Subset constraints of the form described in Section 6.2, and imposed when preventing crashes (Section 6.3) or to cause conditions execute, described above, must be observed.

However, these constraints are not a problem if handled carefully, because in conditional loops, all rows in the data structure are looped through, and the condition only needs to be satisfied once for each loop. Recall from Section 6.2 that the solver which determines command parameters and data structure contents operates as follows:

• Choose a fixed length to use for all of the data structures.

• Determine the constraint graph, where vertices are data structure columns and edges are subset constraints between columns.

• For each connected component, generate a set of random values depending on the length of the data structure.

• For each column (vertex) in each connected component, populate it with a permutation of the set of random values.

When the solver is provided with a temperature $t$, it uses the temperature in the following way:

• The length (i.e., number of tuples) of each data structure is equal to $t$

• Instead of generating a set of random values for each connected component, the same set of random values is used for each column of the same type. Integer values are taken from the set $0, \ldots, t - 1$. String values are taken from some arbitrary set of $t$ distinct strings, e.g. “a0,” “a1,” \ldots, “a{$t - 1$}.” (Each column is still a random permutation of these elements.)

Note that the temperature is required to be an integer. The NERO inference

\footnote{Temperature is only used when needed, as a part of causing constraints to satisfy as described in this section, and as a part of reducing ambiguity, as described in Section 6.6.2.}
algorithm requires data structures to have length at least 2 so loops can be inferred,\(^8\) so the temperature is required to be an integer at least 2.

Note that the probability two values in the same column are equal is 0, and the probability two values of the same type in two different columns are equal is \(1/t\).

There is also a special temperature, zero, which NERO uses to guarantee that conditions execute (discussed below). If NERO tries to decrease the temperature below 2, it is set to 0 instead. At temperature zero:

- All data structures are of length 2
- All integers are either 0 or 1. All strings are \(S\) and \(S_2\), for some arbitrary strings \(S\) and \(S_2\) (e.g. “a0” and “a1”). (All command parameters are 0 or \(S\), depending on type.)

Additionally, the following stipulation is added at temperature zero:

- All permutations are the same (identity) permutation.

Equivalently, at temperature zero, each data structure will contain two rows: one row containing only the values 0 or \(S\), and a second row containing only the values 1 or \(S_2\).

At lower temperatures, because every condition is an equality check, conditions are more likely to satisfy, and in particular, at temperature zero, every condition is guaranteed to satisfy at least once per execution. (See Section 6.8.1 for a discussion on why.) Lower temperatures increase the probability of observing the execution of conditional loop bodies, but also increase ambiguity (See Section 6.4.1) and give less information on rejecting incorrect conditions (Section 6.6). Thus, the temperature must be chosen carefully.

The inference algorithm automatically adjusts the temperature when interacting with the seed program to try to identify the highest temperature (thus reducing

---

\(^8\) The contents of the loop body are inferred by taking the section of the trace between two calls to \_\_next\_, which mark the start of each iteration. If the data structure only has a single element, this algorithm fails.
ambiguity) which allows the condition to satisfy with non-negligible probability.\footnote{In the current implementation of \textsc{Nero}, it is especially important to minimize ambiguity the first time the condition is satisfied, because the set of possible conditions is only generated the first time the condition is satisfied, and because \textsc{Nero} considers arbitrary conjunctions, the number of possible conditions generated is exponential in the amount of ambiguity.}
The inference algorithm begins with an arbitrary large initial temperature. The algorithm slowly decreases the temperature to zero until the unknown condition is satisfied at least once, that is, the iteration executes at least once. When the condition is satisfied, the inference algorithm checks the current value assignments and updates the set of potential conditions. The algorithm may collect multiple potential conditions in two circumstances: (1) there is ambiguity in the data sources for a concrete value (see Section 6.4.1) that potentially participates in the condition and (2) there are multiple pairs of concrete values that equal. In these cases where there are multiple potential conditions, \textsc{Nero} uses another temperature-based method to resolve ambiguity, elaborated on below in Section 6.6.2.

Although the temperature-based algorithm is more versatile than the constraint-based algorithm, it is preferable to use the constraint-based algorithm where possible. The constraint-based algorithm is preferable because if the constraint-based algorithm successfully caused a condition to satisfy, the constraint can be kept to ensure the condition always satisfies at least once each time the loop is executed in the future. In contrast, if the temperature-based algorithm is used, even after the condition is inferred, the constraint solver is not strong or smart enough to impose a constraint forcing the condition to satisfy at least once. Because no constraint forces the condition to satisfy, the temperature must be kept relatively low to perform any inference on statements inside the body of the conditional loop. If there are nested conditional loops where the conditions contain conjunctions, \textsc{Nero} will need to keep the temperature quite low and will need a substantially longer amount of time to infer the contents of the nested loop and to resolve ambiguity in the nested loop.

\footnote{In the current implementation of \textsc{Nero}, it is especially important to minimize ambiguity the first time the condition is satisfied, because the set of possible conditions is only generated the first time the condition is satisfied, and because \textsc{Nero} considers arbitrary conjunctions, the number of possible conditions generated is exponential in the amount of ambiguity.}
6.6.2 Using Temperature to Reduce Ambiguity

NERO uses a temperature pruning method to resolve ambiguity both in conditions as well as more generally in sources of values inferred by the algorithm. It is necessary to use a temperature-based technique for resolving ambiguity in sources of values instead of just executing the seed program repeatedly. A temperature-based technique is necessary because a data structure access with an ambiguous input value source may lie inside a conditional loop which was inferred using the temperature method, so the temperature must be kept low to allow the ambiguous access to occur at all.

The algorithm starts the temperature at a large value and repeatedly halves the temperature until the unknown condition in the seed program is satisfied at least once, or the data structure access with an ambiguous input value source is executed at least once.\textsuperscript{10} Then, the algorithm repeatedly executes the seed program using the same temperature (and different assignments for each execution), until the unknown condition is satisfied again or the data structure access with ambiguous input source is executed again. Each time the condition is satisfied or the data structure access executes, NERO increases the temperature by a small constant factor. Increasing the temperature in this way allows NERO to reduce ambiguity and to eliminate spurious conditions, while still enabling the seed program to enter the conditional block or execute the data structure access. The algorithm repeatedly executes the seed program in this way, until it has finished inferring the condition or until it does not make progress over a large number of executions, at which point it assumes the different possible conditions are semantically equivalent. Since this algorithm adjusts the temperature automatically during inference, it eventually adapts to the seed program.

Once all conditional loops have been explored, all conditions have been inferred, all sequences of function calls (data structure accesses, etc.) in the seed program have been determined, and all (false) ambiguity in conditions and function call input sources have been eliminated, the inference is complete. The inference algorithm then returns its output, discussed next.

\textsuperscript{10}Unlike in the case of inferring an unknown condition, there is no exponential blowup the first time the condition is satisfied or the ambiguous access is executed.
6.7 Inferred Representation

The output of the inference algorithm is an intermediate representation (IR) encoding the seed program’s functionality. In this thesis, I refer to the IR produced by the inference algorithm as the inference IR. The structure of the inference IR closely follows the structure of the domain-specific language supported by NERO (Section 3.1). The domain-specific language allows for an arbitrary sequence of “Steps,” where a Step may be a data structure access, print statement, or a loop over a data structure, possibly with a condition, where the body is a sequence of Steps. The inference IR is structured in the same way, as an arbitrary sequence of function calls (data structure accesses, print statements, and loops). See Figures 2-9,2-10 in the Example chapter for an example.

For each function call, the inference IR indicates

- The type and UID of the data structure accessed (the mapping between names and UIDs is returned separately), or \textbf{print}

- The function call made, e.g. \texttt{getitem} or \texttt{print}

- The sources of the arguments, as locations in the context (see below).

For each argument, the inference IR captures the set of possible locations in the context the argument could have come from. Each context location is represented by a pair, e.g. \((0, 0)\). Recall that the context is a list of tuples. The first number in the context location is the list index; in this case, 0 refers to the 0th element in the list, i.e., the first command parameter. The second number is the tuple index; in this case, 0, because the first command parameter is a single-element tuple. If an argument could possibly come from multiple locations in the context, this is represented in the inference IR as a set containing multiple context locations. NERO tries to resolve context locations unambiguously, but sometimes ambiguity is inherent in the seed program, e.g., if a list is accessed twice with the same index lookup.

For a print statement, the inference IR additionally captures the print statement \textit{template}, which is a Python format string containing placeholders. The placeholders
are filled by values taken from the context. The inference IR records the set of possible context locations for each placeholder in the format string template.

The inference IR represents a loop over a data structure as a head, which indicates the data structure being looped over (type and UID) as well as the condition for the loop, if any, and body, which is a sequence of data structure accesses, print statements, or loops.

In this manner, the inference IR completely captures the functionality of the seed program, and is sufficient (alongside some additional information, such as the database schema and the mapping between UIDs and data structure names, which were determined at the beginning of inference) for regenerating an equivalent program.

6.8 Correctness

To argue the correctness of the inference algorithm, I show (where “function call” is shorthand for “data structure access or print statement or loop”):

1. That the inference algorithm will successfully execute all code paths of the seed program’s commands at least once

2. That the inference algorithm will correctly infer the (nested) sequence of function calls for any part of the seed program which executes

3. That the inference algorithm will correctly determine a superset of possible sources for each input argument to a function call

4. That the inference algorithm will correctly determine a superset of possible conditions for each conditional loop

5. That, with high probability, the inference algorithm will eliminate all false ambiguities in function call input arguments and conditions

Recall from Section 6.7 that the inference algorithm outputs, following the format of the domain-specific language (DSL) supported by Nero (Chapter 3):
• The (nested) sequence of function calls, and
• The sources of each input argument (or condition, for loops) for each function call in the sequence.

Items (1) and (2) together show that the sequence of function calls itself is correct, and (3), (4), and (5) show the sources of each input argument/condition are correct with high probability, which demonstrates the algorithm is correct with high probability. The subsections below argue each of points (1)–(5) semi-formally, so as to capture the intuition behind the algorithm’s correctness while avoiding messy details.

6.8.1 Execution of all code paths

I show that the inference algorithm will successfully execute all code paths of the seed program’s commands at least once.

In the DSL supported by Nero, conditional statements are highly restricted. Therefore, the only variation in code paths is whether conditional loops execute or not, so it suffices to show that each conditional loop executes at least once in this algorithm. But the inference algorithm explicitly guarantees that each conditional loop executes at least once:

If a conditional loop does not execute through other means, the inference algorithm will use the temperature method described in Section 6.6.1, where the temperature will slowly be reduced to 0. When the temperature is 0, each list and each dict will contain some row in which every integer in the row (including the list index, if applicable) is 0, and every string in the row (including the dict key, if applicable) is some arbitrary fixed string $S$. Furthermore, each command parameter will be either 0 or $S$. I now show that under these circumstances, any condition is guaranteed to satisfy at least once in each execution.

I claim that at least once in each execution, at any conditional loop for any seed program, every value of the same type will be equal (to 0, for integers, or $S$, for strings). (Here “value” refers both to values in the context and values retrieved for the current iteration from the data structure being looped over.) Values in the context
can be one of three sources. I claim that, at least once in each execution, none of these sources will introduce values to the context other than 0 and $S$:

1. A value in the context may be a command parameter, which at temperature 0 is either 0 or $S$ depending on type.

2. A value in the context may be retrieved from a data structure using another value in the context as a list index or dict key. However, if the only values in the context are 0 and $S$, then only data structure contents corresponding to index 0 or key $S$ can be retrieved. But the contents for these rows in the data structure, as described above, are all 0 and $S$ at temperature 0.

3. A value in the context may be retrieved by iterating over a data structure. However, values in a loop are not allowed to be accessed or stored outside the loop. Therefore the only values retrieved by iterating which are relevant are values retrieved in the “current” iteration, for any loops that we are currently inside. But the loops we are currently inside will iterate over every row in their data structures. Therefore, when each iteration reaches the row containing only 0 and $S$, all values in the context will be 0 or $S$.

When all values in the context are 0 or $S$, then once the conditional loop in question iterates over the row in the data structure only containing 0 or $S$, because conditions can only compare between values of the same type, all conditions will satisfy and the conditional loop will execute, as desired.

### 6.8.2 Correct inference of sequence on execution

I show that the inference algorithm will correctly infer the (nested) sequence of function calls for any part of the seed program which executes.

In the DSL supported by NERO, conditionals can only be used in a limited fashion. In particular, in any seed program, there are no branching paths (e.g., if there is both a “then” and “else” clause). Instead, conditionals can only prevent the seed program from entering a loop body. The only other “conditional” is when a list or dict lookup
crashes due to an IndexError or KeyError, which halts the execution. In both cases, for any seed program execution, the instrumentation (Chapter 5) directly records each function call (data structure access, etc.) executed. It also clearly indicates where any part of the execution did not occur due to a conditional or crash, by recording an empty loop body or error, respectively.\footnote{As indicated in a footnote in Section 6.6, the trace does not clearly record whether the last iteration of a conditional loop is empty or not. However, as noted in the same footnote, this does not lead to ambiguity; \textsc{Nero} determines whether the last iteration executes or not by comparing the trace with inference results from other executions. Therefore, the lack of explicit indication on whether the last iteration executes or not does not affect my argument, and I do not discuss this in this section.}

Therefore, on each execution the instrumentation will report a subset of the full seed program structure (i.e., a subset of the actual nested sequence of function calls), with a clear placeholder (i.e., an empty loop iteration) where portions of the structure are omitted. The inference algorithm directly reads the structure of the seed program from the trace reported by the instrumentation, leaving a placeholder if a conditional loop has not been inferred yet. Therefore the correct sequence of data structure accesses, print statements, and loops will be inferred for each portion of the seed program that executes.

### 6.8.3 Determining a superset of input argument sources

I show that the inference algorithm will correctly determine a superset of possible sources for each input argument to a function call.

The inference algorithm infers possible input argument sources based on where values in the context match up with values passed into data structure accesses or print statements. A possible source is eliminated only if value in the context from that source does not match up with the value passed into the data structure access. However, in the domain-specific language supported by \textsc{Nero}, the source for each input argument is fixed. (For example, if in one execution a list index operation is performed using a command parameter as an index, for every execution, the same list index operation, if executed, will use the same command parameter as an index, even though the value of the parameter may be different.) Therefore, the correct
source will always have a value in the context matching the value passed into the data structure access and will never be eliminated from consideration. Thus the possible input argument sources considered by the inference algorithm will be a superset of the correct source.

6.8.4 Determining a superset of conditions

I show that the inference algorithm will correctly determine a superset of possible conditions for each conditional loop.

Similar to the case of input argument sources, NERO will consider all possible conditions (of which there are finitely many due to the constraints in the domain-specific language supported by NERO) and eliminate conditions only when a concrete execution trace indicates they are incorrect. No concrete execution trace will indicate the true condition is incorrect, so the true condition will always be one of the conditions considered by NERO.

6.8.5 Eliminating false ambiguity with high probability

I claim that with high probability, the inference algorithm will eliminate all false ambiguities in function call input arguments and conditions. In this section, I will argue only that the inference algorithm will eliminate all false ambiguities in function call input arguments. However, the argument for conditions is similar.

As I will show in my argument, seed programs which are complex in certain adversarial ways can contain ambiguity which has an arbitrarily low probability of being resolved in any given execution, regardless of the temperature. Depending on configuration, NERO will either have an arbitrarily low probability of fully resolving ambiguity in these adversarial cases, or it will take an arbitrarily long time to execute (by increasing the number of executions it tries before assuming the ambiguity is inherent in the seed program).

Because of the existence of adversarial examples, I will not attempt to argue for rigorous guarantees in the probability of resolving false ambiguity. In multiple
places, to simplify my argument, I will disregard unlikely edge cases or assume that probabilities are independent. The assumptions that unlikely edge cases do not occur, or that probabilities are independent, will be stated over the course of my argument. I will also reiterate the assumptions I made at the end of my argument. Despite the assumptions I make, my argument will reveal many important facts about the execution of seed programs in \textsc{Nero}'s domain-specific language (Section 3.1). Ultimately, my argument will show that for seed programs which follow the assumptions I make, \textsc{Nero} can eliminate all false ambiguity with high probability. In practice, \textsc{Nero} was able to eliminate all false ambiguity in the four potential applications discussed in Chapter 8.

Below, I describe the structure of my argument. My argument discusses resolving an ambiguity in the source of a function call input argument. Below, I use $F$ to refer to the function call with an ambiguous input argument source.

1. I first establish preliminary definitions.

2. Next, I establish the relationship between equality of values in the context and equality of values in the original data provided to the seed program.

3. Next, I discuss under what conditions two values in the context will equal. To do so, I will use the relationship established in Step 2.

4. Next, I discuss how temperature (Section 6.6.1) affects the probability that two values in the context are equal. Temperature is important because it is manipulated in the algorithm for resolving ambiguity (Section 6.6.2). In this step, I will use the conditions established in Step 3.

5. Next, I discuss the probability that, at a given temperature, $F$ is reached in the execution. In this step, I will use the results I established in Step 4, because conditions in conditional loops cannot be satisfied without two values being equal.

6. Next, I discuss the probability that, at a given temperature, the ambiguity is resolved. In this step, I will use the results established in Step 4, because
ambiguity cannot be resolved without two values being unequal. I will also use results established in Step 5, because to resolve the ambiguity, $F'$ must execute.

7. Next, I discuss the temperature range chosen by the ambiguity resolution algorithm (Section 6.6.2). I will use the results I established in Step 5, because the ambiguity resolution algorithm selects the temperature based on whether $F$ executes.

8. Finally, I discuss the probability that the ambiguity resolution algorithm is able to successfully resolve the ambiguity. I will use the results I established in Steps 6 and 7.

I begin by establishing preliminary definitions.

Definitions

Here, I establish the preliminary definitions needed in my argument. Throughout the course of my argument, I will establish a few additional definitions, but most definitions I define in advance here.

First, I establish some definitions to clarify what *ambiguity* is.

- **Function call**: As a reminder, I use “function call” to refer to any data structure access or print statement in the seed program.

- **Context location**: As a reminder, a context location is a location in the seed program containing a value available to the seed program. Informally, a context location corresponds to a variable used in the seed program. For example, each parameter is a context location, and each time a tuple of values is retrieved from a list or dict, a context location is defined for each value retrieved. One objective of the inference algorithm is to determine which context locations are the sources for each argument for each function call in the seed program.

- For function call input arguments, ambiguity can arise for a particular input argument of a particular function call. I define:
– **Ambiguous function call:** The function call involved in the ambiguity in the manner defined below

– **Ambiguous input argument:** The input argument of the ambiguous function call involved in the ambiguity, in the manner defined below

Ambiguity arises when, for every execution which reaches the ambiguous function call, there are two locations in the context both containing a value matching the value of the ambiguous input argument. For clarity, I define:

– **Ambiguous context locations:** The two context locations involved in the ambiguity. In every execution observed by the inference algorithm thus far, if the execution reaches the ambiguous function call, the values in the two ambiguous context locations both equal the value of the ambiguous input argument.

• For the sake of completeness, I define ambiguity for conditions as well. For conditions, ambiguity can arise for the condition of a particular conditional loop (“ambiguous conditional loop”). Ambiguity arises when, for every execution which reaches the ambiguous conditional loop, there are two possible conditions which have always been both satisfied when the loop executes, or both unsatisfied when the loop does not execute. (When there are two possible conditions, this can be because a value retrieved from the data structure matches values in two context locations, or two values retrieved from the data structures each match with a context location.) In this discussion, I primarily discuss ambiguity for function calls because a similar argument can be made for ambiguity for conditions.

Next, I establish definitions to clarify the difference between true and false ambiguity.

• **Value assignment:** A value assignment is an assignment of values to command parameters and contents of data structures.

• **Disambiguating value assignment** for an ambiguous function call: For a particular ambiguous function call, a disambiguating value assignment is a
value assignment such that the ambiguous function call is executed but the two ambiguous context locations have different values.\textsuperscript{12}

- **False ambiguity**: A *false ambiguity* is an ambiguity where a disambiguating value assignment exists.

- **True ambiguity**: A *true ambiguity* is an ambiguity where no disambiguating value assignment exists. In other words, for any value assignment, *if the ambiguous function call is executed*, the two ambiguous context locations both have the same value as the ambiguous input argument. In the case of true ambiguity, \textsc{Nero} cannot determine which of the two ambiguous context locations was used as the source for the function call input argument in the seed program. However, \textsc{Nero} can arbitrarily choose one of the ambiguous context locations to use as the source of the function call input argument in the regenerated program; regardless of the choice, the regenerated program will be semantically equivalent to the seed program.

Note that a true ambiguity does not require the two ambiguous context locations to have the same value for *every* value assignment. Instead, they only need to have the same value for value assignments *where the ambiguous function call is executed*. For example, consider a conditional loop where the condition compares values in two context locations \(X\) and \(Y\).\textsuperscript{13} Depending on the value assignment, it is possible \(X\) and \(Y\) will have different values. However, suppose the conditional loop body contains a function \(f\) which uses \(X\) as the source of an input argument. Then there is a true ambiguity in the argument of \(f\), because the condition will not satisfy and \(f\) will not

\textsuperscript{12}Therefore, exactly one of them is equal to the function call input argument. I do not consider the case where the value in neither ambiguous context location matches the function call input argument, because I assume (without loss of generality) that one of the ambiguous context locations is the true source of the function call input argument.

\textsuperscript{13}Strictly speaking, this example is not allowed by the \textsc{Nero} DSL because conditions are only allowed to compare a value retrieved from the data structure being looped over with a value in a context location. However, this example serves to clarify my point. It is also still true that a true ambiguity will result inside such a conditional loop, because the value retrieved from the data structure used in the condition is immediately put into the context. The part of the example which is incorrect is the hypothetical false ambiguity, because the value retrieved from the data structure used in the condition is not actually accessible outside the loop.
execute unless $X$ and $Y$ have the same value.\textsuperscript{14} In contrast, if a function $g$ outside the loop uses $X$ as the source of an input argument, this does not create a true ambiguity, because $g$ can execute even when $X$ and $Y$ are not equal.

Throughout my argument, I will analyze what happens in an execution of the seed program. For clarity of argument, I establish some additional definitions.

- **Executing assignment** for an ambiguous\textsuperscript{15} function call: An *executing assignment* is a value assignment such that the ambiguous function call is reached in the execution. Note that a disambiguating value assignment for an ambiguous function call is, by definition, required to be an executing assignment for the ambiguous function call.

- **Execution point**: An *execution point* is a particular time during the execution of the seed program, specifying the line of code currently being executed as well as progress through all loops in the seed program. (For example, “the second line after the loop has iterated three times.”) It is important to specify progress through loops to fix the value retrieved from the data structure being looped over.

- **Data location**: A *data location* is a particular location in a data structure (as identified by a particular row (list index/dict key) and column) or command parameter. NERO directly controls the initial values stored in each data location when executing the seed program.

Recall that to resolve ambiguity, NERO uses the temperature pruning algorithm described in Section 6.6.2, with temperature described in Section 6.6.1. The temperature pruning argument begins with a high temperature, then repeatedly halves the temperature until an executing assignment occurs. Each time an executing assignment occurs thereafter, the temperature is raised by a small constant factor. I will discuss

\textsuperscript{14}Note that the ambiguity is in the argument to function call $f$, not an ambiguity in the condition of the loop. The condition is not the ambiguity; rather, it creates the ambiguity in $f$.

\textsuperscript{15}The concept applies to any function call, but it is only important in this discussion for ambiguous function calls.
the properties of the temperature pruning algorithm in my argument, so I establish a final set of definitions.

- **Pruning temperature range**: The range of temperatures selected during the slow temperature increase phase in the temperature pruning algorithm, i.e., the range of temperatures selected after the first time an executing assignment occurs.

- **Initial pruning temperature**, or \( t_0 \): The first temperature for which an executing assignment occurred, and the lowest temperature in the pruning temperature range.

- **Executing probability** or \( P(\text{exec} \mid t) \): For a given seed program and a given temperature \( t \), the probability that a value assignment randomly generated at temperature \( t \) is an executing assignment.

- **Initial executing probability**, or \( P(t_0 < t) \): For a given seed program and a given temperature \( t \), the probability that the initial pruning temperature is less than \( t \). Equivalently: during the initial phase of the temperature pruning algorithm, the temperature is repeatedly halved until an executing assignment occurs. \( P(t_0 < t) \) is the probability that an executing assignment did not occur for any temperature greater than \( t \).

- **Disambiguating probability** or \( P(\text{disamb} \mid t) \): For a given seed program and a given temperature \( t \), the probability that a value assignment randomly generated at temperature \( t \) is a disambiguating value assignment.

With definitions established, I begin with the first phase of my argument.

**Context-Data Relationship**

In this phase of my argument, I establish the relationship between context locations and data locations. Establishing the relationship is helpful to my argument because ambiguity occurs when two context locations contain the same value, but value
assignments define what values are assigned to data locations. Establishing the relationship will help with understanding when two context locations will contain the same value.

I claim that at any given execution point, the value stored in any context location was originally one of the values stored in a data location. Each value must originally have been stored in a data location because in the domain-specific language supported by NERO (Section 3.1), none of the allowed operations introduce new values into the program; they can only retrieve values already stored in a data structure.\textsuperscript{16} Therefore, for any given execution, the value \(v\) in each context location originally came from some data location, which was given the value \(v\) in the value assignment. I define this as a term:

\begin{itemize}
  \item **Source data location**: For a context location at a given execution point, the source data location is the data location originally assigned the value stored in the context location.
\end{itemize}

The source data location is unique, for a particular context location at a particular execution point. There are a few things to clarify:

\begin{itemize}
  \item For any given context location, the value stored in it may have originally appeared in multiple data locations. However, in the execution of the seed program, it is possible to trace where each value comes from: e.g., when a value is retrieved from a data structure, that value is retrieved from a particular data location in the data structure. Therefore, the source data location for a context location, as defined by tracing the path of values through the execution in this manner, is still unique. This argument can be formalized using induction.
\end{itemize}

\textsuperscript{16}Strictly speaking, this is not true. There is one way to introduce a new value, which is to perform a list append operation: then the new list index may be a new value if it is not already stored in a data structure or command parameter. In most cases, the introduction of the new list index value will not prevent NERO from eliminating false ambiguity. Additionally, it is unusual for seed programs to access the new list index. One reason is that seed programs typically read from data structures, then perform all write operations at the end. Another reason is that the new list index can only be accessed by iterating through the list, which will also access many other elements. It is rare to single out the new list index, because it cannot be accessed individually using a list index lookup. Because my analysis is not meant to provide rigorous guarantees, and because it is rare for the new list index to affect my analysis, I ignore the introduction of new values in my analysis.
• The source data location for a context location may be different from execution to execution, or at different execution points within the same execution (because of iteration). For example, consider a list index lookup operation. This lookup operation creates a context location storing the retrieved value. The source data location will depend on what the list index is. However, for any particular execution, the list index will be a concrete value, and it will be possible to identify a single source data location.

• Values may be stored into data structures during an execution. If a value in a data structure has been overwritten, then retrieved again, the source data location is the data location which contained the value originally. For example, suppose I retrieve value \(a\) from list \(l\), then store \(a\) into list \(l_2\). I then retrieve value \(b\) from list \(l_2\). The source data location for \(a\) is one of the data locations in \(l\). Usually, the source data location for \(b\) is one of the data locations in \(l_2\). However, if \(b\) was actually retrieved from where \(a\) was stored into, then the source data location for \(b\) is actually the same as the source data location for \(a\), i.e., one of the data locations in \(l\).

With source data locations defined, I next discuss what causes two context locations to contain the same value.

**Value Equality in Context Locations**

Here, I discuss under what conditions two context locations may contain the same value.

Each context location contains the same value as the value originally stored in their source data location. Consider two ambiguous context locations \(C_1\) and \(C_2\), and let their source data locations be \(S_1\) and \(S_2\). Then \(C_1\) and \(C_2\) can contain the same value \(v\) if:

1. \(S_1\) and \(S_2\) are different data locations which both were assigned the value \(v\) originally, or
2. $S_1$ and $S_2$ are the same data location, $S$. This case is possible if:

(a) The $C_1$ and $C_2$ are retrieved from the same row and same column of the same data structure, or

(b) For one or both context locations $C_1$ and $C_2$, the value from $S$ was stored into a data structure, then retrieved into the context location.

To simplify this analysis, I assume case 2 is only possible if $C_1$ and $C_2$ are retrieved from the same column of the same data structure. In case 2a, this assumption is true by definition. In case 2b, this assumption does not hold, because only one of $C_1$ or $C_2$ is required to be retrieved from a data structure; however, case 2b does not typically occur in seed programs. There is also a situation equivalent to case 2b, where the same-column same-structure assumption does hold. Suppose $S$ is the data location corresponding to a command parameter, and $C_1$ is the context location containing the same command parameter. Suppose that $S$ is stored into a list $l$ under index $i$. Then, a value is retrieved from $l$ using index $j$, stored into $C_2$. If $i == j$, then this scenario falls under case 2b. Then this is equivalent to the situation where, instead of $C_1$ being the context location containing a command parameter, $C_1$ is defined by retrieving $S$ immediately into $C_1$ using the list index $i$. In this second scenario, $C_1$ and $C_2$ are retrieved from the same column of the same data structure. However, $C_1$ is guaranteed to be the same as $S$, which makes the two scenarios equivalent.

When $C_1$ and $C_2$ share a value via case 2, it is typically because two other context locations $C_1'$ and $C_2'$ also share the same value $v'$, which I call the recursive scenario. For example, suppose $C_1$ and $C_2$ are both retrieved by list index operations on the same data structure. Then case 2a can occur if the list indices used to retrieve $C_1$ and $C_2$ are sourced from context locations $C_1'$ and $C_2'$ containing the same value.

In the recursive scenario, $C_1'$ and $C_2'$ must fall into one of the two cases described above. If $C_1'$ and $C_2'$ are from the same column of the same data structure, then $C_1'$ and $C_2'$ might contain equal values via case 2, in a recursive manner. Otherwise, $C_1'$ and $C_2'$ can only contain equal values via Case 1.17

---

17It is possible $C_1'$ and $C_2'$ are actually the same context location, for example if the same command
There are also other ways for Case 2 to occur other than the recursive scenario, but they behave similarly to the recursive scenario so I do not explicitly discuss them in my analysis. For example, \( C_1 \) may be retrieved by looping over a list, and \( C_2 \) may be retrieved by performing a list index operation with a parameter \( P \). Then case 2 may occur when the iteration index \( i \) of the loop equals \( P \), so that the loop and the list index operation access the same row of the list. However, there is a second scenario which is equivalent (for the purposes of my analysis), and which qualifies as a recursive scenario. Suppose instead of retrieving \( C_1 \) in a loop, there is a new parameter \( Q \), and \( C_1 \) is retrieved using \( Q \) as a list index. Then if \( Q \) equals \( i \), and \( i = P \), then again \( C_1 \) and \( C_2 \) access the same data location, so the new scenario is equivalent to the old one. However, the new scenario qualifies as a recursive scenario. Therefore, in my analysis I assume Case 2 only occurs due to the recursive scenario.

In summary, depending on the structure of the seed program, there are multiple ways for the context locations \( C_1 \) and \( C_2 \) to contain the same value \( v \). The two context locations might contain the same value via case 1, where the source data locations \( S_1 \) and \( S_2 \) share the value \( v \). Alternatively, if \( C_1 \) and \( C_2 \) are retrieved from the same column of the same data structure, the recursive scenario (or an equivalent scenario) may occur. In the recursive scenario, there may be two context locations \( C'_1 \) and \( C'_2 \), whose source locations \( S'_1 \) and \( S'_2 \) share a value \( v' \). If \( C'_1 \) and \( C'_2 \) are themselves retrieved from the same column of the same data structure, the recursive scenario can occur again. The number of times the recursive scenario can occur depends on the seed program.

To make this more concrete:

- **Chain**: For two context locations \( C_1 \) and \( C_2 \), a chain ending at \( C_1 \) and \( C_2 \) is a sequence of pairs of context locations \( C_{1,i} \) and \( C_{2,i} \), with \( i \in 1, \ldots, m \), such that:
  
  - For \( 2 \leq i \leq m \), \( C_{1,i} \) and \( C_{2,i} \) are retrieved from the same column of the same data structure.

  parameter is used as a list index to retrieve both \( C_1 \) and \( C_2 \). However, if \( C'_1 \) and \( C'_2 \) are the same context location, then \( C_1 \) and \( C_2 \) are forced to be equal in a true ambiguity, so I disregard this case.
- For $2 \leq i \leq m$, and $j \in \{1, 2\}$, $C_{j,i}$ is retrieved using $C_{j,i-1}$ (e.g., as a list index or dict key, or in a conditional loop comparing to $C_{j,i-1}$).
- $C_{j,m} = C_j$, for $j \in \{1, 2\}$.

Informally, a chain is two parallel sequences of data structure accesses, where each access in each sequence uses the result of the previous. Note that for any two context locations $C_1$ and $C_2$, there exists a chain ending at $C_1$ and $C_2$: the chain of length 1 with $C_{1,1} = C_1, C_{2,1} = C_2$.

**Chain length:** For two context locations $C_1$ and $C_2$, the *chain length* is the length of the longest chain ending at $C_1$ and $C_2$. Note that the chain length is always at least 1, because there always exists a chain of length 1, as remarked upon in the definition of a chain.

Then if the chain length of $C_1$ and $C_2$ is $m + 1$, the recursive scenario can occur $m$ times. In this case, $C_1$ and $C_2$ will share the same value if the source data locations of any $C_{1,i}$ and $C_{2,i}$ contain the same value.

I conclude that $C_1$ and $C_2$ can share a value for the following reason:

- Let $C_1$ and $C_2$ be two context locations, and $m$ be the chain length of $C_1$ and $C_2$. Then there are $m$ pairs of source data locations $S_i$ and $S'_i$, such that if *any* $S_i$ and $S'_i$ were originally assigned the same value, $C_1$ and $C_2$ will share the same value. (The pairs of source data locations may overlap, i.e., the same data location may occur in multiple pairs.) Each pair $S_i, S'_i$ are the source data locations of some pair of context locations in the longest chain ending at $C_1$ and $C_2$.

As an intuitive example: suppose $C_1$ and $C_2$ are each the end of a sequence of list index operations. Multiple times, a value is retrieved from a list, then used as the index for the next list index operation. Then if the retrieved values for the two sequences happen to be equal at any point, the remaining portion of both sequences will be the same, and $C_1$ and $C_2$ will have the same value.
Having established the reason for two context locations sharing the same value, I now discuss how temperature affects the probability that two context locations share a value in a given value assignment.

**Effects of Temperature on Equality**

In this section, I discuss how temperature affects whether values in two context locations are equal or not.

Recall from Section 6.6.1 that at any nonzero temperature $t$, the following effects occur:

- The number of rows in each data structure is equal to $t$
- There are $t$ possible values selected for each data location, selected uniformly at random.

Therefore, for any two data locations $S_1$ and $S_2$ in different columns of the same type, the probability $S_1$ and $S_2$ share the same value is equal to $1/t$.

Let $C_1$ and $C_2$ be two context locations, and let $m$ denote their chain length. Recall that there exist $m$ pairs of data locations $S_i$ and $S_i'$ (each corresponding to a pair of context locations on the longest chain ending at $C_1$ and $C_2$), such that if any pair $S_i$ and $S_i'$ share the same value, $C_1$ and $C_2$ will share also share the same value.

For this analysis, I make the following independence assumption:

- **Data Location Equality Independence (DLEI)** assumption: For two different pairs of data locations $(S_i, S_i')$ and $(S_j, S_j')$, the probability that $S_i$ and $S_i'$ share the same value is independent of the probability that $S_j$ and $S_j'$ share the same value.

The DLEI assumption holds if no more than two of $S_i, S_i', S_j, S_j'$ are from the same column, because values in different columns are generated independently. However, if $S_i$ and $S_j$ are from the same column and $S_i'$ and $S_j'$ are also from the same column, then the DLEI assumption does not hold, because values in a single column are not generated independently of each other.
However, in my analysis, I make the DLEI assumption, because my analysis is intended to be an estimate as opposed to a precise calculation. The objective of my analysis is to derive a formula to estimate the probability that Nero will find a disambiguating value assignment through repeated execution. This formula will indicate that for seed programs encountered in practice, Nero is able to find a disambiguating value assignment with high probability. When the DLEI assumption does not hold, the final probability estimate may be incorrect. But for seed programs encountered in practice, the probability estimate will only be off by a small factor, on the order of 5 or less.\textsuperscript{18} In this case, however, Nero can still find a disambiguating value assignment with high probability by increasing the number of repeated executions. The DLEI assumption dramatically simplifies my analysis, and does not affect my conclusion of the types of seed programs that Nero can find a disambiguating value assignment for with high probability. Therefore, I make the DLEI assumption.

Let $v_1$ denote the value in $C_1$ and let $v_2$ denote the value in $C_2$, for a given execution point and a given value assignment. Let $m$ denote the chain length of $C_1$ and $C_2$. Then at a temperature $t$, the probability $v_1 = v_2$ is, assuming DLEI:

$$P(v_1 = v_2 \mid t) = 1 - \left(1 - \frac{1}{t}\right)^m \tag{6.1}$$

Note that the minimum possible probability is when $m = 1$,\textsuperscript{19} therefore

$$P(v_1 = v_2 \mid t) \geq \frac{1}{t} \tag{6.2}$$

Now that the probability of value equality at a given temperature has been established, I discuss the probability that a value assignment selected at a given

\textsuperscript{18}I estimate the error from assuming DLEI in the following calculation: The final probability estimate will be a product of different factors. When DLEI (or one of the independence assumptions I will make in a later part of my analysis) does not hold, each individual factor may be slightly incorrect, e.g., a factor might be $\frac{1}{2}$ instead of $\frac{2}{3}$, which is an error of 0.75. (This error comes because values in a single column are not generated independently of each other, and columns will be length 3 for the final calculation. Then a value may be sampled without replacement, leading to a collision probability of $\frac{1}{2}$ instead of $\frac{1}{3}$.) There are a total of five factors in the final probability expression. Stacking up the errors together is an error factor of $0.75^5 \approx 0.24$, which approximately a factor of 4, but I round up to a factor of 5 to be conservative.

\textsuperscript{19}Recall that the chain length is always at least 1.
temperature is an executing assignment.

**Probability of an Executing Assignment at a Given Temperature**

I now discuss the probability that a value assignment selected at a given temperature is an executing assignment.

_A priori_, for a given ambiguous function call $F$, a value assignment can fail to be an executing assignment for two reasons:

1. The program crashes due to a KeyError or IndexError before the function call is executed.

2. The condition of a conditional loop containing the ambiguous function call is not satisfied.

For case 1, recall that NERO generates constraints to prevent KeyErrors and IndexErrors from occurring (Section 6.3), so Case 1 will not occur.

For case 2, recall that when NERO tries to cause conditions to satisfy, it first tries a constraint-based methodology (Section 6.6.1), in which NERO tries imposing different constraints to cause conditions to satisfy. Let $C$ denote the set of constraints imposed by NERO. Then some conditions are forced to satisfy by constraints in $C$. However, it is possible the seed program contains conditions which are not forced to satisfy by constraints in $C$, because NERO does not impose constraints to force conjunctions of equality checks to satisfy. Therefore, case 2 may prevent a value assignment from being an executing assignment.

At a given temperature, what is the probability that conditions in conditional loops prevent the ambiguous function call $F$ from executing? If $F$ is located in one or more nested conditional loops, all conditions in the nested loops must be satisfied for $F$ to execute. In other words: there exists a set of equality checks $\mathcal{E}$, such that $F$ will execute if there exists an execution point of $F$ such that every equality check in $\mathcal{E}$ is satisfied. $\mathcal{E}$ is the set of equality checks in conditions of the conditional loops which contain $F$. 

123
However, recall that NERO tries to impose constraints to cause conditions to satisfy (Section 6.6.1). Again, let \( \mathcal{C} \) denote the set of constraints imposed by NERO. Then some of the equality checks in \( \mathcal{E} \) may be forced to satisfy because of some constraint in \( \mathcal{C} \). The remaining equality checks can only be satisfied when the randomly generated value assignment happens to cause them to satisfy. I give a name to the equality checks which are not forced to satisfy:

- **Unconstrained condition:** For an ambiguous function call \( F \), an unconstrained condition is an equality check in \( \mathcal{E} \) which is *not* forced to satisfy by any constraint in \( \mathcal{C} \). In other words, an unconstrained condition is an equality check in a conditional loop containing \( F \), such that the condition is not forced to satisfy by any constraint. More informally: an unconstrained condition is a *condition* which is not *constrained* to satisfy. (An equality check in \( \mathcal{E} \) which is not an unconstrained condition is a *constrained condition*.)

As noted, unconstrained conditions can only be satisfied when the randomly generated value assignment happens to cause them to satisfy; constrained conditions are forced to satisfy by constraints. Therefore, to determine the probability that a value assignment generated at a given temperature is an executing assignment depends only on unconstrained conditions.

What is the probability that, at a given temperature, an unconstrained condition is satisfied? Equation 6.2 showed the probability that two context locations have the same value. For a given (unconstrained) condition, let \( C_1 \) and \( C_2 \) denote the context locations corresponding to the values compared.

In a condition, one of the values being compared was retrieved from a data structure in a loop (Chapter 3), so it is not retrieved using another value as a list index. Therefore, the chain length of \( C_1 \) and \( C_2 \) is at most 2. However, it is rare for seed programs to use a condition to compare two values from the same column of the same data structure. (Usually, a value from a data structure is compared with a command parameter, or with a value retrieved from a different data structure.) Therefore, the chain length of \( C_1 \) and \( C_2 \) is typically 1. In this analysis, I focus on the scenario
where the chain length of $C_1$ and $C_2$ is always 1. (If the chain length is 2 instead of 1, the probability estimates I derive will be incorrect by a small factor. This error will affect my final estimate of the probability that NERO can find a disambiguating value assignment. However, the probability change does not affect my final conclusion that NERO can find a disambiguating value assignment with high probability through repeated execution. If NERO has a lower probability of generating a disambiguating value assignment for a single execution, NERO can still find a disambiguating value assignment with high probability by increasing the number of repeated executions.)

Assuming the chain length is always 1 and applying Equation 6.1, the probability that any given unconstrained condition holds is $t^{-1}$.

To compute the probability that every unconstrained condition is satisfied simultaneously, I make the following independence assumption:

- **Unconstrained Condition Satisfaction Independence (UCSI) assumption:** the satisfaction of any unconstrained condition is independent of the satisfaction of any other unconstrained condition.

My reasoning for the UCSI assumption is similar to my reasoning for the DLEI assumption. The UCSI assumption is similar to the DLEI assumption because both assume the independence of equality in pairs of values. A condition compares values in two context locations $C$ and $C'$, and the condition is satisfied if the source data locations of $C$ and $C'$ share a value. Therefore, the UCSI assumption holds in a situation analogous to when the DLEI assumption holds: when conditions are not comparing values from the same pair of columns. Similar to DLEI, the UCSI assumption simplifies my analysis and does not affect my final conclusion regarding the seed programs that NERO can find a disambiguating value assignment for with high probability. Therefore, I make the UCSI assumption.

With the UCSI assumption, it is straightforward to compute the probability that every unconstrained condition is satisfied simultaneously:

- For a given ambiguous function call $F$, let $n$ be the number of unconstrained
conditions needed for $F$ to execute.\textsuperscript{20} Then, at a given temperature $t$, the probability that a randomly generated value assignment is an executing assignment for $F$ is (assuming UCSI)

$$P(\text{exec} \mid t) = t^{-n}$$  \hspace{1cm} (6.3)

In this analysis, I focus on the $n \geq 1$ case, for the following reason: because when $n = 0$, every (obeying the constraints imposed by the inference algorithm) is an executing assignment, so executing assignments occur at arbitrarily high temperatures. As I show below, because of the arbitrarily high temperatures, when $n = 0$, disambiguating value assignments can occur with probability close to 1. Therefore, I focus on the $n \geq 1$ case, where NERO will have more difficulty discovering a disambiguating value assignment.

Next, I discuss the probability at a given temperature that a randomly generated value assignment is a disambiguating value assignment.

**Probability of a Disambiguating Value Assignment at a Given Temperature**

I now discuss the probability that a randomly generated value assignment selected at a given temperature is disambiguating value assignment.

For a given ambiguous function call $F$, let $C_1$ and $C_2$ be the two ambiguous context locations. A disambiguating value assignment is a value assignment which:

1. is an executing assignment for $F$, such that

2. $C_1$ and $C_2$ contain different values.

In the discussion of the probability of an executing assignment, I showed that an executing assignment is a value assignment where certain values must equal. In the discussion of the effects of temperature on value equality, I showed that $C_1$ and $C_2$ are different when certain values (namely, values in pairs of data locations) are not equal.\textsuperscript{20}\footnotetext{Note that a conditional loop containing a conjunction of $k$ unconstrained conditions is counted $k$ times.}
In some cases, the equality requirements from (1) conflict with the disequality requirements for (2). When such a conflict occurs, it is impossible for \( C_1 \) and \( C_2 \) to contain different values when \( F \) executes, and the ambiguity is a true ambiguity. As I noted in the definition for true ambiguity, Nero can correctly infer and regenerate the functionality of a seed program without resolving true ambiguities; Nero will conclude \( C_1 \) and \( C_2 \) are equivalent, and will choose one to use arbitrarily. My analysis discusses when Nero can eliminate false ambiguity. Therefore, my analysis focuses on the case where equality requirements for (1) do not conflict with disequality requirements for (2).

To proceed with my analysis, I make one more independence assumption:

- **Execution-Disequality Independence (EDI):** (1) and (2) above occur independently of each other. In other words, a randomly generated value assignment is an executing assignment independent of the value assignment causing \( C_1 \) and \( C_2 \) to contain different values.

The EDI assumption is similar to the DLEI and UCSI assumptions. All three assumptions hold in similar situations (i.e., when values from different columns are compared), and I make all three assumptions for similar reasons: because it simplifies my analysis, and my final conclusion (as discussed in my justification of the DLEI assumption) is unchanged when EDI, DLEI, or UCSI do not hold. However, there is one notable difference between EDI and the other two assumptions: when the requirements for (1) conflict with the requirements for (2), the EDI assumption is false and a true ambiguity occurs. As noted above, I focus on the case where a true ambiguity does not occur. When the requirements for (1) do not conflict with the requirements for (2), the EDI assumption sometimes still does not hold (for similar reasons to why DLEI or UCSI sometimes do not hold), but the consequences are similar to when DLEI or UCSI do not hold. Therefore, in the case my analysis focuses on, where requirements for (1) and (2) do not conflict, my conclusion is unaffected when EDI does not hold. Therefore, in the case my analysis focuses on, it is reasonable for me to assume EDI, much as I have assumed DLEI and UCSI.
To derive the probability that a randomly generated value assignment is a disambiguating value assignment, we begin with:

\[ P(\text{disamb} \mid t) = P(\text{exec} \mid t) \cdot P(\text{disamb} \mid \text{exec}, t) \]

where \( P(\text{disamb} \mid \text{exec}, t) \) is the probability that a value assignment \( \mathcal{A} \) generated at temperature \( t \) is a disambiguating value assignment, conditioned on \( \mathcal{A} \) being an executing assignment.

Let \( v_1 \) and \( v_2 \) denote the values in \( C_1 \) and \( C_2 \). Then applying EDI, we have

\[ P(\text{disamb} \mid t) = P(\text{exec} \mid t) \cdot P(v_1 \neq v_2 \mid t) \]

Then, applying Equations 6.1 and 6.3, we derive the following result:

- For a given ambiguous function call \( F \), let \( n \) be the number of unconstrained conditions needed for \( F \) to execute. Let the ambiguous context locations be denoted \( C_1 \) and \( C_2 \), and let \( m \) denote their chain length. Then at a given temperature \( t \), the probability that the value assignment generated is a disambiguating value assignment for \( F \) is (assuming DLEI, UCSI, and EDI):

\[ P(\text{disamb} \mid t) = t^{-n} \left( 1 - \frac{1}{t} \right)^m \]  

(6.4)

- Taking the derivative of this equation with respect to \( t \), we find that the probability the value assignment generated is a disambiguating value assignment for \( F \) is maximized at

\[ \arg \max_t P(\text{disamb} \mid t) = \frac{n + m}{n} \]  

(6.5)

- Plugging this in, the maximum probability that a disambiguating value assignment is generated is

\[ \max_t P(\text{disamb} \mid t) = \frac{n^m m^m}{(n + m)^{n+m}} \]  

(6.6)
(Note that for \( n = 0 \), this optimum temperature is unbounded and the optimum probability is 1, which is why I assumed \( n \geq 1 \) previously.)

Having established the probability of a disambiguating value assignment, I discuss the pruning temperature range.

**Pruning Temperature Range**

Here, I discuss what pruning temperature range is selected by the temperature pruning algorithm defined in Section 6.6.2.

Because the temperature pruning algorithm searches for a temperature where executing assignments occur, it is possible to bound the initial executing probability \( P(t_0 < t) \) for a given temperature \( t \), where \( t_0 \) denotes the initial pruning temperature. Recall that the temperature pruning algorithm halves the temperature until an executing assignment occurs. Let \( t_{\text{start}} \) be the initial (high) temperature of the temperature pruning algorithm. Assume without loss of generality that there exists \( k \) such that \( t_{\text{start}} = t \cdot 2^k \). Then \( P(t_0 < t) \) is the probability that none of the value assignments generated at temperatures \( t \cdot 2^k, t \cdot 2^{k-1}, \ldots, t \) are executing assignments.

Using a union bound:

\[
P(t_0 < t) \geq 1 - \sum_{i=0}^{k} P(\text{exec} \mid t \cdot 2^i)
\]

Using Equation 6.3 and assuming \( n \geq 1 \), where \( n \) is the number of unconstrained conditions which must satisfy, Equation 6.7 becomes a geometric series, giving

\[
P(t_0 < t) \geq 1 - \sum_{i=0}^{k} t^{-n} \cdot 2^{-in} > 1 - 2t^{-n}
\]

For a given temperature \( t \), what is the probability that the pruning temperature range will include \( t \)? As long as \( t_0 < t \), if the temperature pruning process continues for sufficiently long, and the temperature is raised sufficiently quickly, the temperature \( t \) will eventually be reached. If \( t_0 \) is unusually low, \( t \) will take a longer time to be reached. However, at lower temperatures, executing assignments will occur more
frequently, so the temperature will increase more quickly. Therefore, if $t_0$ is unusually low, it will not take significantly longer for the temperature to reach $t$. To simplify the analysis, I will assume that $t$ is in the pruning temperature range as long as $t_0 < t$; this assumption will hold as long as $t$ is not too high. Therefore:

- For an ambiguous function call $F$, let $n$ be the number of unconstrained conditions needed for $F$ to execute. For any $t > 0$ not too large, $t$ is in the pruning temperature range with probability at least

$$1 - 2t^{-n} \quad (6.9)$$

Finally, I give an estimate for the probability that NERO is able to eliminate false ambiguity.

**Probability False Ambiguity is Eliminated**

In this section, I give an estimate for the probability that NERO is able to find a disambiguating value assignment for a given ambiguous function call, assuming the ambiguity is false.

It is difficult to directly compute the probability that NERO is able to find a disambiguating value assignment, because a range of different temperatures are tested in the temperature pruning algorithm, and these temperatures are not determined beforehand. Instead, I perform a more heuristical analysis.

For an ambiguous function call $F$ with ambiguous context locations $C_1$ and $C_2$, let $n$ be the number of unconstrained conditions needed for $F$ to execute, and let $m$ be the chain length of $C_1$ and $C_2$.

Equation 6.6 gives the maximum probability (with the maximum taken over possible temperatures) that a random value assignment is a disambiguating value assignment. It is clear from Equation 6.6 that as $m$ and $n$ increase, the probability of finding a disambiguating value assignment decreases exponentially. In theory, NERO could still find disambiguating value assignments by performing many executions at the optimum temperature for disambiguating value assignments. In this case, NERO can
still find disambiguating value assignments with high probability, but with expected runtime exponential in \( m \) and \( n \).

However, most seed programs have small \( m \) and \( n \). While seed programs may perform long sequences of data structure accesses, it is rare for seed programs to perform two long sequences of data structure accesses in parallel, where the two sequences access the same columns of the same data structures in the same order. Therefore, many seed programs have \( m = 1 \), and most seed programs have \( m \leq 2 \). (For comparison, all of the benchmark commands (Chapter 8) have \( m = 1 \).) Similarly, while seed programs may have many nested conditional loops, nested conditional loops usually use simple equality checks for conditions, not conjunctions. Conjunctions are uncommon because nested conditional loops typically correspond to \texttt{JOIN} operations, which typically only join on a single column at once. Conditional loops using simple equality checks for conditions can be caused to execute using the constraint-based methodology discussed in Section 6.6.1. Therefore, they do not qualify as unconstrained conditions, and do not contribute towards \( n \). Consequently, most seed programs have \( n \leq 3 \). (For comparison, most of the benchmark commands (Chapter 8) have \( n = 0 \), a few have \( n = 1 \), and two have \( n = 2 \).)

Equation 6.6 indicates that disambiguating value assignments occur more rarely when \( m \) and \( n \) are both large. To be conservative, I estimate the probability that \textsc{Nero} is able to find a disambiguating value assignment for \( m = 2, n = 3 \).

Even with \( m \) and \( n \) fixed, it is difficult to directly compute the probability that a value assignment is a disambiguating value assignment. Instead, I perform the following estimation procedure:

1. Find a temperature \( t_{0.9} \) such that with probability at least 0.9, \( t_{0.9} \) is in the pruning temperature range.

2. Assume \textsc{Nero} executes the seed program \( k \) times at \( t_{0.9} \), for some \( k \).

3. Find the probability that a disambiguating value assignment is discovered in this process, in terms of \( k \).
I justify this procedure as follows: Equation 6.4 shows that the probability that a given execution uses a disambiguating value assignment decreases monotonically as the temperature increases past (or decreases past) $\frac{m+n}{n} = \frac{5}{3}$ (Equation 6.5). Nero will most likely execute the seed program several times with temperature less than $t_{0.9}$. Because of the monotonicity property of Equation 6.4, each of these executions at $t < t_{0.9}$ will discover a disambiguating value assignment with higher probability than performing each execution at $t_{0.9}$. The proposed estimation procedure assumes executions occur at $t_{0.9}$, therefore in many cases it will underestimate the probability of discovering a disambiguating value assignment. Therefore, if the proposed estimation procedure indicates the probability of discovering a disambiguating value assignment is high, then Nero can be considered (heuristically) to resolve false ambiguity with high probability.

First, $t_{0.9}$ solves the following equation, derived from Equation 6.9 with $n = 3$:

$$0.9 \leq 1 - 2t_{0.9}^{-3}$$

Therefore,

$$t_{0.9} \geq 0.05^{-\frac{1}{3}} \approx 2.714$$

Temperature is required to be an integer, so I round up and set

$$t_{0.9} = 3$$

Plugging into Equation 6.4 with $m = 2, n = 3$, the probability that a value assignment generated at $t = t_{0.9}$ is a disambiguating value assignment is:

$$3^{-3} \left(1 - \frac{1}{3}\right)^2 \approx 0.01646$$

Therefore, the probability that a disambiguating value assignment is found with $k$ executions is

$$1 - (1 - 0.01646)^k \approx 1 - 0.98354^k$$
For \( k = 300 \) executions, this value is approximately 0.993, which shows that NERO will resolve false ambiguity with high probability for \( m \leq 2, n \leq 3 \). (In comparison, the NERO prototype implementation tries 600 executions before stopping if the ambiguity is not resolved.)

**Ambiguity Elimination Conclusion**

The assumptions I made over the course of this argument were as follows:

- Seed programs do not introduce new values to the context which were not present in data structures or command parameters at the beginning of the execution. As discussed in a footnote in the discussion on the Context-Data Relationship, it is only possible for seed programs to introduce new values by performing a list append operation and then performing a conditional loop over the list to access the new list index.

- No single command in the seed program stores a value into a data structure and subsequently retrieves the same value. In other words, case 2b in the discussion on Value Equality in Context Locations does not occur.

- For any false ambiguity which arises in the seed program between two ambiguous context locations \( C_1 \) and \( C_2 \), and for any two pairs of source data locations \((S_i, S_i')\) and \((S_j, S_j')\) of context locations in any chain ending at \( C_1 \) and \( C_2 \), the probability that \( S_i \) and \( S_i' \) share the same value is independent of the probability that \( S_j \) and \( S_j' \) share the same value. This is the DLEI assumption from the discussion on the Effects of Temperature on Equality, applied to the source data locations of context location pairs in chains ending at \( C_1 \) and \( C_2 \).

- For any false ambiguity which arises in the seed program for any function call \( F \), for any two unconstrained conditions \( U_1 \) and \( U_2 \) required for \( F \) to execute, satisfaction of \( U_1 \) occurs independently of satisfaction of \( U_2 \). This is the UCSI assumption from the discussion on the Probability of an Executing Assignment.
at a Given Temperature, applied to the unconstrained conditions required for $F$ to execute.

- For any false ambiguity which arises in the seed program for any function call $F$ and ambiguous context locations $C_1$ and $C_2$, a randomly generated value assignment is an executing assignment for $F$ independent of the value assignment causing $C_1$ and $C_2$ to contain different values.$^{21}$

- For any false ambiguity which arises in the seed program for any function call $F$ and ambiguous context locations $C_1$ and $C_2$, the chain length of $C_1$ and $C_2$ is at most two and the number of unconstrained conditions for $F$ is at most three.

Seed programs can in general make it exponentially difficult$^{22}$ for NERO to resolve false ambiguity if they contain many unconstrained conditions or have context location pairs with a long chain length. However, for seed programs which satisfy the assumptions above, NERO can resolve all false ambiguity with high probability.

### 6.8.6 Correctness Analysis Summary

I have shown that the NERO inference algorithm will always infer the correct structure of function calls in the seed program; and that NERO will always consider the true source of each input to each function call or condition to be a possibility; and that with high probability, NERO will eliminate ambiguity and find each true source in the seed program (though for complex programs it may take a longer time or have a lower probability (depending on NERO’s configuration), especially for adversarial programs). I conclude that the NERO inference algorithm will, in all practical cases, correctly infer the functionality of seed programs conforming to the domain-specific language (DSL) supported by NERO.

---

$^{21}$Note that this assumption does not require independence of execution and value equality in cases of true ambiguity.

$^{22}$i.e., the probability of eliminating false ambiguity is exponentially small for a fixed number of repeated executions, or the runtime is exponentially long for a fixed probability of eliminating false ambiguity.
As demonstrated throughout my argument, correctness of this algorithm depends heavily on the DSL supported by Nero. Important aspects of this DSL include:

- A fixed structure of accesses/print/loops (with limited use of conditionals), where on each execution a subset of the true structure is observed with a clear placeholder for missing components
- Conditionals structured in a way that the inference algorithm can guarantee full execution
- The source of the input arguments to each function call in the seed program are fixed locations in the context (e.g., stored in a fixed variable, in contrast to e.g., if ternary operators were allowed)
- Values are only ever compared for equality, limiting the number of different possible distinguishable value assignments.

After Nero successfully infers the functionality of the seed program, it returns a regenerated program implementing the same functionality. I discuss the regeneration algorithm next.
Chapter 7

Regeneration for Database Backend

After the NERO inference algorithm produces an intermediate representation (IR) for the functionality of each seed program command (Section 6.7), the NERO regeneration algorithm produces a new *regenerated program* which implements the inferred functionality but uses a relational database to store data instead of using Python built-in data structures. The regenerated program is a Python program which uses SQL queries to access a database, with the *sqlite3* Python package. The regenerated program contains:

- Boilerplate for generate database tables and access the database
- A command line interface similar to the command loop NERO generates to augment the seed program prior to inference (Section 4.1)
- Python functions implementing all the inferred commands
- Handlers allowing the user the populate the database tables with values stored in a file, similar to the fill command handlers NERO generates to augment the seed program prior to inference (Section 4.2)

Note there is no fundamental reason the regenerated program must be a Python program accessing a SQLite database. The regenerated program could theoretically be implemented in any language which supports databases, and a more complex database (such as a distributed database) could be used. To regenerate to a different language
or to use a more complex database, one would simply write a new regeneration module compatible with the IR returned by the inference algorithm (the *inference IR*). The prototype implementation of NERO regenerates to Python and SQLite as a simple proof-of-concept. A regeneration module for another language or database or is straightforward to implement, but would take time to implement, and would add little to this thesis.

Each Python function in the regenerated program implements a command by making a sequence of SQL queries. Intermediate results are stored in a list, the *context*. The regeneration algorithm can be modified to store intermediate results in variables instead for readability purposes. In this case, the variable names can be systematically generated based on the name of each database table and column.

In Section 7.1, I describe the regeneration algorithm. In Section 7.2, I describe the behavior of the regenerated program.

## 7.1 Regeneration Algorithm

To produce a regenerated program, the NERO regeneration algorithm first determines the database schema for the regenerated program (Section 7.1.1), then regenerates each command to use the database (Section 7.1.2), and finally generates the remaining boilerplate (Section 7.1.3).

Section 2.4 in the Example chapter gives an example of how NERO uses the regeneration algorithm to produce a regenerated program for the student registration application.

### 7.1.1 Regenerating the Database Schema

NERO first determines the database schema for the regenerated program. The schema is constructed in a straightforward manner using information in the data specification file written by the user (Section 4.2). The data specification file directly specifies the database schema to be used in the regenerated program, including the name of each table and the name and type of each column in each table. Each database table has a
direct correspondence with a single data structure in the seed program, as indicated
in the data specification file. For more details on the data specification file and how
it specifies the database schema and the correspondence with data structures in the
seed program, refer to Section 4.2.

7.1.2 Regenerating Inferred Commands

To regenerate each command, NERO begins with the inference IR produced by
the inference algorithm, described in Section 6.7. The regeneration algorithm is
straightforward, because the regenerated code closely follows the inference IR. First,
the inference IR is converted to a second intermediate representation, the SQL IR.
Then, NERO performs optimizations on the SQL IR. Finally, NERO outputs a Python
function corresponding to the SQL IR. Below, I explain each step of the regeneration
algorithm in detail.

To generate the SQL IR, NERO preserves the nested structure of the inference
IR and converts each data structure access in the inference IR in a 1:1 manner to a
SQL query. Each SQL query accesses the database table corresponding to the data
structure accessed in the inference IR (see Section 4.2 for more information on the
correspondence). Each access corresponds to a SQL query as follows:

- A list index lookup or dict key lookup is replaced by a SELECT query with a
  WHERE clause conditioning on the primary key of the corresponding database
table

- A list index store is replaced by a UPDATE query conditioning on the primary
  key of the corresponding table

- A list append is replaced by a INSERT INTO query

- A dict key store is replaced by a INSERT OR REPLACE INTO query

- A print statement is left as a print statement
For loops, recall from Section 6.7 that a loop in the inference IR has two parts: a head (`__iter__`) and body (sequence of commands starting with `__next__`). The head is replaced by a `SELECT` query which retrieves all rows of the corresponding table, or if the loop has a condition, retrieves all rows matching the condition using a `WHERE` clause. The `__next__` in the body is skipped, and the rest of the body is placed in a `for` loop iterating over the return value of the head, and converted to the SQL IR in a recursive manner.

Recall from Section 6.4 that NERO conceptualizes the seed program as using a context to store values accessible to the seed program. Recall from Section 6.7 that the input to each function call in the inference IR is represented as a context location, a pair which indicates a particular location in the context. The SQL IR likewise uses context locations to indicate the parameters for each SQL query. For example, a list index lookup in the inference IR uses a context location to indicate what value in the context is used as the list index. The corresponding query in the SQL IR is a `SELECT` query with a `WHERE` clause which compares the integer primary key of the table with a lookup parameter. The SQL IR uses a context location to indicate what value in the context is used as the lookup parameter.

After converting the inference IR to the SQL IR, NERO performs optimizations on the SQL IR. NERO performs two related optimizations on the SQL IR, both of which simplify loops:

- Let $L_1$ denote a loop (conditional or otherwise) in the SQL IR. Recall that each loop in the SQL IR is composed of a head, which retrieves the data in the table, and a body, which loops over the data and performs operations on it. The head of $L_1$ is a `SELECT` query which retrieves rows of a database table $T_1$, and the body of $L_1$ iterates over each row retrieved by the head. Suppose the body of $L_1$ consists of a conditional loop $L_2$ over a database table $T_2$. If the condition of $L_2$ compares a value retrieved from $T_2$ with a value retrieved from $T_1$ by the head of $L_1$, NERO will combine $L_1$ and $L_2$ into a single loop with a `JOIN` operation:

  - The head of the new loop is a `SELECT` query which `JOINs` $T_1$ and $T_2$ on the
condition of \( L_2 \).

- The body of the new loop is the original body of \( L_2 \), plus some boilerplate to ensure that context locations remain intact.\(^1\) (Note that this optimization procedure can only be performed if the body of \( L_1 \) does not contain any operations other than \( L_2 \).)

In this manner, NERO replaces two nested loops in the SQL IR with a single loop using an efficient JOIN operation.

- Let \( L_1 \) denote a loop (conditional or otherwise) over \( T_2 \). Suppose the first operation in the body of \( L_1 \) is a SELECT query \( Q \) on table \( T_2 \), with a WHERE clause specifying the primary key of \( T_2 \) should equal a value retrieved from \( T_1 \) by the head of \( L_1 \). (In other words, this operation corresponds to a list index or dict key lookup operation, using a value retrieved by the loop as the index or key.) Then NERO merges \( Q \) into the head of \( L_1 \), in the following manner:

  - The head of \( L_1 \) is replaced by a SELECT query which JOINs \( T_1 \) and \( T_2 \) on the same condition used by the WHERE clause of \( Q \), i.e., the primary key of \( T_2 \) is compared with some column of \( T_1 \).

  - \( Q \) is removed from the body of the loop, and is replaced by boilerplate to ensure context locations remain intact. Namely: the values from \( T_2 \) retrieved by the new head of the loop are copied into the context at the previous location of \( Q \).\(^2\)

Note that because the JOIN operation is performed on the primary key of \( T_2 \), which contains unique values, the new loop performs the same number of iterations as the original \( L_1 \). Using this optimization, NERO is able to simplify the body of the regenerated loop.

It is possible to perform other optimizations on the SQL IR, but these are the two

---

\(^1\)This boilerplate can be optimized away.

\(^2\)Again, this can be optimized away.
optimizations included in the prototype NERO implementation as a proof of concept. I suggest additional optimizations in Section 9.5.

The prototype implementation of NERO performs an additional adjustment to the SQL IR to make the regenerated program compatible with the seed program. In particular, list indices in Python are 0-indexed, but integer primary keys in SQLite are 1-indexed. The regenerated program accesses integer primary keys everywhere the seed program accesses list indices. The regenerated program subtracts 1 from the value of each integer primary key before accessing the key. Subtracting 1 from each integer primary key allows the regenerated program to use the same input parameters as the seed program, and to output the same values, so that it is easy to automatically test whether the regenerated program accurately reproduces the functionality of the seed program. It is straightforward to disable this feature if the user desires.

After optimizing the SQL IR for each command, NERO converts the SQL IR into a Python function (accessing a SQLite database) which implements the SQL IR. Each operation in the SQL IR corresponds to a snippet of code in the regenerated Python function, and the nested structure of the SQL IR is preserved using Python for loops.

The regenerated program operates by performing a sequence of SQL queries and storing the result of each SQL query in a list, the context. The regenerated program follows the structure of the SQL IR. Values are added to the context as follows:

- Most SELECT queries performed by the regenerated program retrieve a single row of a database table, because the SELECT query has a WHERE clause conditioning on the primary key of the table, which contains unique values. These SELECT queries correspond to list index and dict key lookup operations in the seed program. The result of each of these SELECT queries is stored as a tuple in the context, where each position in the tuple corresponds to a column of the database table.

- The remaining SELECT queries are the head of a loop, and correspond to looping over a list or dict in the seed program. These queries may return multiple rows of a database table, and the result is stored as a list of tuples in the context.
Each tuple corresponds to a single row of the database table, and each position in each tuple corresponds to a column of the database table. The body of the loop iterates over the result of the head.

- Queries other than SELECT (such as UPDATE, etc.) do not retrieve any values. A None value is left in the context as a placeholder, so that each query corresponds to a single entry in the context. These placeholders can be optimized out.

Each query determines its input parameters by accessing values in the context. The value accessed in the context by each query is determined by the context locations stored in the SQL IR for the parameters of that query.

The regenerated program outputs values by printing them to the console, in the same manner as the seed program.

The regenerated program additionally contains automatically generated input validation. The seed program may contain list index or dict key lookup operations which can crash due to an IndexError or KeyError if a requested index or key is not found. In the regenerated program, list index or dict key lookup operations correspond to SELECT queries which retrieve the row of a database table containing a selected primary key value. If no row in the table contains the requested primary key value, the regenerated program will gracefully abort the command and report an error message. More complex error handling can be added in future work, but is not necessary for the proof-of-concept provided by NERO as described in this paper.

### 7.1.3 Regenerating Boilerplate

The regenerated program produced by NERO includes boilerplate in addition to the Python functions implementing each command. The boilerplate includes:

- Boilerplate to connect to the SQLite database at the entry point of the regenerated program.

- Boilerplate to generate database tables. Immediately after connecting to the SQLite database, the regenerated program executes a CREATE TABLE IF NOT
exists query for each database table used by the regenerated program.

• A command line interface. Similar to the command loop Nero generates to augment the seed program prior to inference (Section 4.1), the command line interface prompts the user for an input, then parses the input and dispatches to the appropriate Python function. The prompt lists the set of commands available to the user, as well as the types of each input parameter for each command. If the user enters an invalid command, an error message is printed. After each command is executed, the regenerated program commits the transaction to the database.

• Handlers allowing the user to repopulate database tables with fresh values. Similar to the fill command handlers Nero generates to augment the seed program prior to inference (Section 4.2), each fill command handler takes a file name as an input parameter, then overwrites the contents of the corresponding database table with the data stored in the file. Fill command handlers are accessible using the command line interface.

• Boilerplate to disconnect from the SQLite database when the user exits the regenerated program.

With the inclusion of this boilerplate, the regenerated program is a fully executable Python program the user can interact with. The regenerated program’s behavior is described in Section 7.2.

7.2 Regenerated Program Behavior

The regenerated program differs from the seed program in how it stores data, but the regenerated program otherwise implements the same functionality as the seed program.\(^3\) The regenerated program uses a database to store data, while the seed

\(^3\)Without the indexing adjustment described in Section 7.1.2, there would be an additional difference, in that each primary key in the regenerated program will be off-by-one from the corresponding list index in the seed program. The regenerated program will still be able to execute, but all corresponding data and inputs would need to be incremented by one.
program uses Python data structures. The two programs produce the same behavior on the same data, except data in the regenerated program persists across multiple executions.

The seed program initializes a set of empty data structures when the program is loaded. Without using the fill command handlers generated by Nero, commands would not have access to any initial data in the data structures. If the user executes the seed program, quits the program, and launches it again, the user will find the seed program has reverted to its initial empty state.

In contrast, because the regenerated program uses a database to store data, data persists between executions. If the user quits the regenerated program and launches it again, the behavior of the program will be the same as if the user had not quit the program.

The persistence of data is an advantage of using databases instead of using data structures. By using Nero, the user is able to utilize the advantages of databases without needing to know how to implement a database program themselves. Similarly, user could use Nero to utilize the advantages of languages other than Python, or of distributed databases, etc. Utilizing these advantages would require only that a regeneration module be written for Nero for the new language, distributed database, etc. The new regeneration module would be compatible with the inference IR, and can be written in a straightforward manner by using the same process as the regeneration algorithm described in this chapter.
Chapter 8

Experimental Results

I implemented seed programs for four potential applications that may benefit from using a database. Each seed program takes commands as input, stores and retrieves data from the Python global data structures, and prints outputs. The applications include:

- A task management application, inspired by the open source Kanban task management application [4].

- A chat room application, inspired by the open source Kandan chat room application [5].

- A blog application, inspired by Rails example [3].

- A student registration application that I presented in Section 2.

For each of these seed programs, Nero is able to fully infer the functionality and produce a regenerated program. I present each application in the sections below. Then, I discuss how I evaluated Nero’s performance, and discuss the implications of these experimental results.
8.1 Task management application

The first application is a task management application, inspired by the open source Kanban task management application. Below, I give an overview of the task management application, then describe each of the commands in more detail. Because the task management application seed program is too long to fit in a single page, I do not present the entire seed program, but rather present the implementation of each command separately in the Appendix. NERO is able to fully infer and regenerate this application in 5 minutes and 5 seconds.

Users may use the task management application to organize the work they need to do for a project. The task management application stores the following information:

- The application tracks boards. Each board groups together tasks in a related set of projects. The application tracks the users who are registered members of each board.

- Each board contains a number of lists, which contain the tasks for a particular user project.

- Each list contains a number of cards, which each represent a single task for a project.

- Each card can have card comments, which give more details on the task represented by the card.

To store this information, the seed program contains the following data structures:

- The users list. Each user has an ID (represented by the list index) as well as a name, email, and password (each stored as a string).

- The boards list. Each board has an ID (represented by the list index) and a name (stored as a string).

- The lists list. Each list has an ID (represented by the list index) and the ID of the board containing the list (stored as an integer).
Figure 8-1: Data specification file for the task management application. Each entry in this file specifies one of the data structures defined in the text.

- The **cards** list. Each card has an ID (represented by the list index), a title (stored as a string) and the ID of the list containing the card (stored as an integer).

- The **card_comments** list. Each comment has an ID (represented by the list index), content (stored as a string), and the ID of the card containing the comment (stored as an integer).

- The **board_members** list. The **board_members** list contains pairs of integers, representing the ID of a board and the ID of a user who is a member of the board. The list index of each pair is unused.

Figure 8-1 presents the data specification for the file for the task management application, defining the six data structures discussed above. **NERO** regenerates the database schema presented in Figure 8-2.

Figure 8-3 presents the command specification file for the task management application. There are four commands:

- **get_api_board**: This command accepts a user’s email and password as inputs. The command first verifies the user email matches the password. Then, for each board the user is a member of, the command retrieves and outputs information on every list, card, and comment in the board.
CREATE TABLE IF NOT EXISTS Cards (id INTEGER PRIMARY KEY, title TEXT, list_id INT)
CREATE TABLE IF NOT EXISTS Comments (id INTEGER PRIMARY KEY, content TEXT, card_id INT)
CREATE TABLE IF NOT EXISTS Boards (id INTEGER PRIMARY KEY, name TEXT)
CREATE TABLE IF NOT EXISTS BoardMembers (idx INTEGER PRIMARY KEY, board_id INT, member_id INT)
CREATE TABLE IF NOT EXISTS Users (id INTEGER PRIMARY KEY, name TEXT, email TEXT, password TEXT)
CREATE TABLE IF NOT EXISTS Lists (id INTEGER PRIMARY KEY, board_id INT)

Figure 8-2: Regenerated database schema for the task management application. This figure presents the SQLite code used to define each of the tables in the regenerated program. Each line represents one of the database tables defined in Figure 8-1. The data structures corresponding to each database table are indicated in the text.

```
[ "get_api_board", "get_api_board", "str", "str"],
[ "get_api_cards_id", "get_api_cards_id", "str", "str", "int"],
[ "post_api_lists", "post_api_lists", "str", "str", "int"],
[ "post_api_cards", "post_api_cards", "str", "str", "str", "int"],
```

Figure 8-3: Command specification file for the task management application. Each line in the file describes a separate command. I describe each command in the text.
• **get_api_cards_id**: This command accepts a user’s email and password, as well as the ID of a card, as inputs. If the email and password match, and if the user is a member of the board containing the card, this command outputs information on the board and list containing the card, as well as information on each comment contained in the card.

• **post_api_lists**: This command accepts a user’s email and password as well as the ID of a board. If the email and password match, and the user is a member of the board, this command creates a new empty list for the board.

• **post_api_cards**: This command accepts a user’s email and password as well as the ID of a list and the title of a new card to be created. If the email and password match, and the user is a member of the board containing the list, this command creates a new card with the specified title on the specified list.

The seed program could additionally be implemented with four additional commands which add new users, boards, comments, or board members. However, each of these commands would be similar to **post_api_lists** and **post_api_cards**, and do not add any value to this experimental result. Therefore, I have not included these commands in the seed program.

I discuss each of these commands below. All these commands are written using basic Python language constructs. NERO automatically infers all these commands and regenerates more sophisticated implementations for them, using the database backend.

### 8.1.1 get_api_board

The first command, **get_api_board**, first checks the user identity using inputs **email** and **password**. If such a user is found, the command checks the user privilege in table **board_members**. For each board the user is a member of, the command finds all lists in the board, finds all cards in each list, and finds all comments for each card. The command then prints all the retrieved data. Figure A-1 presents the Python function implementing this command in the seed program.
This seed program command contains five nested loops. The outermost loop iterates over the `users` list. The loop body executes only when the conditions `user_email == email` and `user_password == password` both hold, enforcing that an user must exist with the specified email and password. The second loop iterates over the `board_members` list. The loop body executes only when the condition `member_id == user_id` holds, enforcing that the specified user must be a member of the board.

The next three nested loops together imitate a chain of SQL JOIN operations using foreign keys between tables. The third loop iterates over the `lists` list. The loop body executes only when the board ID that list belongs to matches the previously retrieved board ID. The fourth loop iterates over the `cards` list. The loop body executes only when the list ID that the card belongs to matches the previously retrieved list ID. The last loop iterates over the `comments` list. The loop body executes only when the list ID that the card belongs to matches the previously retrieved card ID. Finally, for each comment, the seed program prints the retrieved data. As a result, this command outputs all the boards that the user is a member of, including all the lists in each board, all the cards in each list, and all the comments for each card.

When performing inference on this command, Nero uses the temperature method discussed in Section 6.6.1 to cause the conjunction of two equality checks in the outermost loop, over `users`, to satisfy. Nero is able to cause the remaining conditions to satisfy using the constraint-based methodology discussed in Section 6.6.1, by imposing constraints which force the conditions to satisfy. Note that inside the second loop, the seed program performs a list index operation on the `boards` list. During inference, this causes an IndexError to occur. Nero detects the IndexError and recovers, imposing a constraint to prevent the IndexError from occurring in the future, as discussed in Section 6.3.

Nero successfully infers this command and regenerates new code that uses SQL operations on database tables instead of loops over data structures as in the seed program. As discussed in Section 7.1.2, each of the nested loops is first regenerated into a loop over the result of a `SELECT` query. Nero then optimizes these loops, combining
several nested loops together using a INNER JOIN operation, as shown below.

The regenerated command first performs the SQL query:

```sql
SELECT Users.id - 1, Users.name, Users.email, Users.password,
      BoardMembers.idx - 1, BoardMembers.board_id, BoardMembers.
      member_id, Boards.id - 1, Boards.name FROM Users INNER JOIN
      BoardMembers ON BoardMembers.member_id = Users.id - 1 INNER JOIN
      Boards ON Boards.id - 1 = BoardMembers.board_id WHERE Users.email
      = :lookup0 AND Users.password = :lookup1
```

where strings :lookup0 and :lookup1 are placeholders for SQL library to safely fill in parameters into the query. As I discuss in Section 9.6, these placeholder names are straightforward to replace with more understandable names in future work, but replacing placeholder names with more understandable names does not add to the proof-of-concept provided by Nero as described in this thesis. In this program, these two placeholders are filled in with the two input parameters, email and password. This SQL query contains an INNER JOIN operation on database tables Users, BoardMembers, and Boards, returning the boards that the specified user is a member of. The query also contains a WHERE operation which enforces that the user email and password match in the Users table. The arithmetic that decreases ID columns by one is due to the different indexing conventions for Python and SQL, as discussed in Section 7.1.2.

After performing this query, the regenerated command loops over each row of the retrieved data. For each row, the regenerated program performs the SQL query:

```sql
SELECT Lists.id - 1, Lists.board_id, Cards.id - 1, Cards.title,
      Cards.list_id, Comments.id - 1, Comments.content, Comments.
      card_id FROM Lists INNER JOIN Cards ON Cards.list_id = Lists.id -
      1 INNER JOIN Comments ON Comments.card_id = Cards.id - 1 WHERE
      Lists.board_id = :lookup
```

where string :lookup is a placed-holder for filling in a parameter. This parameter is filled in with the board ID from the row retrieved by the previous SQL query.

As these regenerated SQL queries demonstrate, Nero uses efficient SELECT, JOIN, and WHERE operations to express the inferred data accesses, nested loops,
and conditional loops from the seed program. Figure A-2 presents the regenerated code for this command.

8.1.2 get_api_cards_id

Figure A-3 presents the seed program command get_api_cards_id that retrieves card information after checking user identity, user privilege, and checking card existence in lists and boards.

The get_api_cards_id seed program implementation differs from the seed program implementation of get_api_board in two major ways.

1. The get_api_cards_id implementation uses list index operations to mimic most of the foreign key lookup operations, whereas get_api_board uses conditional loops. Nero is able to infer the sequence of list index operations using the algorithm described in Section 6.4. Each list index operation causes an IndexError the first time it executes, but Nero detects the error and recovers by imposing a constraint preventing the error from occurring again, as discussed in Section 6.3.

2. The get_api_cards_id implementation contains two nested conditional loops whose conditions contain conjunctions of equality checks. Recall from Section 6.6.1 that conditions without conjunctions can be inferred by imposing constraints, but conditions containing conjunctions can only be caused to satisfy by reducing the temperature. Nero is able to infer the nested loops containing conjunctions by reducing the temperature so it is likely for both conditions to satisfy.

The regenerated program uses SQL queries to replace data structures accesses in the seed program. Figure A-4 presents the regenerated code for this command.

8.1.3 post_api_lists and post_api_cards

Figure A-5 presents seed program commands that create a new list in the specified board and that creates a new card in the specified list. Nero infers both commands
in a manner similar to the get_api_board and get_api_cards_id commands. The regenerated program uses SQL queries to replace data structures accesses in the seed program. Figure A-6 presents the regenerated code for these commands.

Next, I discuss the next application, a chat room application.

8.2 Chat room application

The second application is a chat room application, inspired by the open source Kandan chat room application. In this application, users are able to post content in different channels. Figure A-7 presents the seed program for a chat room application. NERO is able to fully infer and regenerate this application in 54 seconds.

This application stores the following information:

- The users list: Stores the user ID (represented by the list index), username (a string), email (a string), password (a string), first name (a string), and last name (a string) for each user.

- The activities list: Each activity is a post by a user on a channel. For each activity, this list stores the activity ID (the list index), the ID of the user who posted (an integer), the ID of the channel posted to (an integer), and the content of the post (a string).

- The channels list: A channel is a place where a user can post on a particular topic. Each channel stores the channel ID (the list index), the name of the channel (a string), and the ID of the user who owns the channel (an integer).

Figure 8-4 presents the data specification file for the chat room application, defining to the three data structures discussed above. NERO regenerates the database schema presented in Figure 8-5.

Figure 8-6 presents the command specification file for the chat room application. I describe the commands below. NERO is able to infer and regenerate these commands in a similar manner to how it inferred and regenerated commands for the task management
"activities": [
  ["int", "int", "str"],
  ["activity_id", "user_id", "channel_id", "content"],
  "Activities"],
"users": [
  ["str", "str", "str", "str", "str"],
  ["user_id", "username", "email", "password", "firstname", "lastname"],
  "Users"],
"channels": [
  ["str", "int"],
  ["channel_id", "name", "user_id"],
  "Channels"]
}

Figure 8-4: Data specification file for the chat room application. Each entry in this file specifies one of the data structures defined in the text.

1 CREATE TABLE IF NOT EXISTS Channels (channel_id INTEGER PRIMARY KEY, name TEXT, user_id INT)
2 CREATE TABLE IF NOT EXISTS Users (user_id INTEGER PRIMARY KEY, username TEXT, email TEXT, password TEXT, firstname TEXT, lastname TEXT)
3 CREATE TABLE IF NOT EXISTS Activities (activity_id INTEGER PRIMARY KEY, user_id INT, channel_id INT, content TEXT)

Figure 8-5: Regenerated database schema for the chat room application. This figure presents the SQLite code used to define the tables in the regenerated program. Each line represents one of the database tables defined in Figure 8-4. The data structures corresponding to each database table are indicated in the text.

["get_channels", "do_get_channels", "str", "str"],
["post", "do_post", "str", "str", "str", "int"]

Figure 8-6: Command specification file for the chat room application. Each line in the file describes a separate command. I describe each command in the text.
application (Section 8.1). Therefore, I do not go into detail on how NERO infers and regenerates commands for the chat room application.

The first command, `get_channels`, first checks the user identity using inputs `username_query` and `password_query`. If such a user is found, the command prints the user information and goes on to loop over all channels. For each channel, it prints the channel information and then prints all chat activities, including their author information, that belong to this channel. This seed program command contains four nested loops. NERO successfully infers this command and regenerates new code that uses SQL operations on database tables instead of loops over data structures. Figure A-8 presents the regenerated code for this command.

The second command, `post`, checks user identity and then creates a new chat activity in the specified channel. The regenerated program uses SQL queries to replace data structures accesses in the seed program. Figure A-9 presents the regenerated code for these commands.

As with the task management application, this chat room application’s commands are written using basic Python language constructs. NERO automatically infers both commands and regenerates more sophisticated implementations for them, using the database backend.

### 8.3 Blog and student registration applications

Below, I briefly describe two additional applications whose functionality is inferred and regenerated by NERO. Because these examples are similar to the task management application (Section 8.1) and chat room application (Section 8.2) and are inferred and regenerated in a similar manner, I do not go into detail.

Figure A-10 presents seed program commands for the blog application. The first command prints all articles. The second command prints the specified article, along with all comments for this article. The third command creates a new article. The last command creates a new comment for an article. NERO successfully infers all commands and regenerates more sophisticated implementations for them, using the database backend.
database backend. Figure A-11 presents the regenerated code for these commands. Nero is able to fully infer and regenerate the blog application in 0.7 seconds.

The student registration application is the example I presented in Section 2. Nero successfully infers both commands and regenerates more sophisticated implementations for them, using the database backend. Figure A-12 presents the regenerated code for these commands. Nero is able to fully infer and regenerate the student registration application in 3 seconds.

8.4 Evaluation

I used a two-fold approach to verify Nero successfully regenerated a program implementing the same functionality as the seed program.

- Automated testing: for each seed program, I wrote a set of sample input commands which thoroughly explore the functionality of the seed program. The automatic tester verifies that the seed and regenerated programs produce the same output.\(^1\)

- Manual inspection: the example programs are relatively short. I read the code of the regenerated program and compared it to the code of the seed program, and verified that the functionality implemented was the same.

Using these two approaches, I was able to verify Nero correctly inferred and regenerated the functionality of each seed program.

8.5 Discussion

These experimental results demonstrate how a user can implement functionality as a seed program:

\(^1\)There is some nondeterminism in the output, because iterations over dicts can happen in any order, and SQL `SELECT` queries can return database rows in any order. The automatic tester accounts for this.
• When the user wishes to represent data canonically represented by an integer (such as an ID), the user may use a list, and the integer ID will be represented by the list index. If instead the user wishes to represent data canonically represented by a string, the user may use a dict. Because using IDs is common practice for database programs, the supermajority of data structures in the applications presented are list data structures. List data structures are also able to replicate the SQL behavior of automatically generating integer IDs.

• When values in data structures refer to values in other data structures, for example where a registration record may store the ID of a student, these values can be accessed either by using a conditional loop or by performing a list index or dict key lookup operation.

• When the user needs to store information, it is possible to do so using a simple list append, list index assign, or dict key assign operation.

The experimental results presented in this chapter give a range of different applications, and NERO is able to successfully infer the functionality of each. This success demonstrates that the seed programs supported by NERO encapsulate a wide range of common use cases.

The experimental results also demonstrate the NERO inference algorithm’s ability to infer the functionality of complex seed programs. Of the four seed programs presented in this chapter, the task management application was the most complex, so I focus on the task management application in this discussion.

The task management application presented an example of a program containing a complex sequence of data structure accesses. The user is able to implement this program using a sequence of nested loops or data structure accesses in Python. NERO is able to successfully infer the functionality of the seed program and produce a regenerated program implementing the same functionality but using a database instead of data structures.

In particular, NERO’s inference of the task management application demonstrates:
• NERO is able to successfully infer seed programs containing deeply nested loops or long sequences of data structure accesses.

• NERO is able to apply constraints to infer equality check conditions in nested loops (Section 6.6.1).

• NERO is able to apply the temperature method to infer conjunctions of equality check conditions in nested loops (Section 6.6.1).

• NERO is able to detect and recover from IndexErrors in the middle of an execution, including the inside of a conditional loop.

For each of the four seed programs presented in this chapter, NERO’s inference algorithm is able to correctly and unambiguously infer the functionality of each seed program without reading the source code. NERO’s regeneration algorithm is able to produce a regenerated program which faithfully reproduces the functionality implemented by the seed program. This success demonstrates NERO’s practical applicability.
Chapter 9

Future work

NERO is able to infer the functionality of seed programs which use a wide range of common computational patterns. After performing inference on such a seed program, NERO produces a regenerated program implementing the same behavior as that of the seed program, but which stores data in a database instead of in data structures. I argued in Section 6.8 that NERO will correctly infer the seed program’s functionality with high probability. The experimental results in Chapter 8 demonstrate the success of NERO’s inference and regeneration algorithms on practical programs.

Nevertheless, it is possible to extend NERO’s functionality further. In this chapter, I discuss suggested future work that can be done to further increase NERO’s flexibility, accuracy, or usability. Some of these extensions are straightforward and can be implemented relatively easily. Others require more time to implement or involve fundamental changes to the NERO inference algorithm.

Each of the following sections discusses a suggested improvement to NERO. The sections are roughly sorted in increasing order of complexity. For example, Section 9.1 describes a simple and superficial change to increase the user-friendliness of NERO. In contrast, Section 9.10 describes a fundamental change to the NERO inference algorithm.
9.1 Interface Improvements

In this section, I discuss possible improvements to the user interface of NERO. Currently, NERO requires the user to write a seed program and write two specification files (Chapter 4).

One improvement which can be made in this area is to increase the human-readability and human-writability of the two specification files. As discussed in Chapter 4, the prototype NERO implementation expects both of these files in JSON format for ease of parsing. However, it is straightforward to implement a parser which can read a file in a more human-readable or human-writable format.

Another improvement which can be made is to improve error detection and reporting. In the prototype NERO implementation, if a specification file contains a formatting error, or if the seed program contains a bug (e.g., a syntax error), NERO may crash. It would be straightforward for NERO to instead gracefully exit after printing an error message describing the problem.

Both of these suggested improvements are straightforward to implement. However, these two suggested improvements only increase the user-friendliness of NERO without improving its functionality. The prototype implementation of NERO is intended as a proof-of-concept. Therefore, these two suggested improvements do not add value to this thesis and have not been included in NERO as described in this thesis.

9.2 Dict String Keys

As noted in Section 3.1.1, NERO requires dicts keys to be strings. Because the prototype NERO implementation is intended to be a proof-of-concept, the use of integer keys in dicts does not add value to this thesis. Therefore, for simplicity, NERO as described in this thesis requires dict keys to be strings. As I describe below, however, it is straightforward to remove the requirement that dict keys be of string type.

Recall that NERO generates fill command handlers which populate data structures in the seed program with specified data (Section 4.2). As explained in a footnote in
Section 4.2, the reason for the string key requirement is that fill command handlers populate data structures using data stored in a file in JSON format. However, keys in JSON objects (which are used to represent dicts) can only be strings. If a dict contained integer keys, the NERO inference algorithm could not store data for the dict in JSON format, to be loaded into the seed program.

NERO could remove the requirement that dict keys be strings in a straightforward manner by modifying the fill command handlers generated for dicts with integer keys. Recall that NERO generates fill command handlers based on a data specification file written by the user (Section 4.2). NERO can learn from the data specification file whether each dict contains integer or string keys. When NERO generates a fill command handler for a dict, NERO could generate a modified fill command handler if the dict key is of integer type. After reading data for the dict stored in JSON format, the modified fill command handler would cast each key to an integer. Then, when the NERO inference algorithm wants to populate an integer-key dict with data, the algorithm can store the data in JSON format by first casting each key in the data from an integer to a string.\footnote{Casting keys to strings is performed automatically by the \texttt{json} Python module used to covert to JSON format.} When the modified fill command handler reads the data, it will automatically cast the keys back to integers.

By allowing dict keys to be integers, NERO will allow seed programs to encode data canonically represented by an integer even if the integers are not all consecutive (as would be required if the data is stored in a list, because list indices are consecutive). For example, consider the student registration application from Chapter 2. If the user wanted to give each student an integer ID in a particular format (e.g. MIT uses nine-digit student IDs), it may be appropriate for the user to store students in a dict with the student ID as an integer key.
9.3 Dict items()

As noted in Section 3.1, seed programs can iterate over data structures to access each entry of the data structure in sequence. However, when a seed program iterates over a dict, the loop only returns the key in each iteration. To access the values in the dict, the seed program must separately perform a dict key lookup operation. This process would be simplified if NERO allowed seed programs to access the items() method of a dict. When iterating over a dict accessed by items(), the loop will return both the key and the value in each iteration.

It is straightforward to modify NERO to support usage of the items() method in seed programs. The method items() returns a dict_items object, which the loop then iterates over. To support the items() method, the NERO instrumentation should be modified to record the items() call. The returned dict_items object should be instrumented as well,\(^2\) to record each time the loop over the dict_items object advances an iteration.\(^3\) Additionally, the inference algorithm should be modified so when it sees the items() function call in the trace, it knows the loop has access to both the keys and values of the dict. These are the only changes which need to be made; except for the value accessed each iteration, iterating with items() is no different from iterating over the dict directly.

When NERO supports the items() method, seed programs which iterate over dicts will be simpler to implement. More importantly, supporting items() will expand the range of conditional loops available to seed programs which iterate over dicts. As specified in Section 3.1, conditional loops must compare a value retrieved by the loop against a value retrieved outside of the loop. When iterating over a dict, only the key can be compared to values retrieved outside of the loop, because only the key is retrieved by the loop. When iterating over a dict items(), any value in each row of the dict can be used in the comparison.

\(^2\)This instrumentation can be done in the same way that iterators returned by __iter__ are instrumented; see Chapter 5.

\(^3\)When Python loops over the dict_items object, it does not call the __next__ function of the dict, unlike usage of the enumerate function for lists. Therefore the dict_items object must be instrumented separately.
The `items()` method of a dict is analogous to the built-in Python `enumerate` function for a list, which is currently supported by NERO (Section 3.1). The reason `enumerate` is supported but `items()` is not is that dicts are used only rarely in seed programs. As discussed in Section 8.5, the rarity of dicts is because it is common practice when writing database programs to assign an automatically-generated integer ID for each object stored in the database. It is appropriate to use a list index as an automatically-generated integer ID. Additionally, modifying NERO to support `items()` is straightforward (as I discussed above) and does not add anything to the proof-of-concept provided by the prototype NERO implementation. Therefore, I did not prioritize supporting `items()` in the prototype implementation.

### 9.4 Value Return

NERO produces a regenerated program which outputs values by printing them to the console. As discussed in Section 3.2, a user may want to use a regenerated program as a module in a more complex program. In this case, the regenerated program would need to implement Python functions which return values instead of printing them.

As I discussed in Section 3.2, it is straightforward to modify the regeneration algorithm to produce a program which returns values instead of printing them. Whenever the original regenerated program would print a set of values, the modified regenerated program can instead append the values to an output list. The regenerated program can then return the output list at the end of the program. The user would still implement a seed program which prints values; only the regenerated program needs to be changed.

The suggested modification would allow the user to regenerate Python functions which return a desired set of values. For example, suppose the user wishes to regenerate a command which returns all students in a database table whose first name matches a given command parameter. The user could implement a seed program which performs a conditional loop over a list of students. If the student’s first name matches the command parameter, the seed program will print out each of the values stored for
the student (the student’s name, ID, etc.), where each value is passed to the print function as a separate argument. After NERO infers the functionality of this seed program, NERO will regenerate a Python function which performs a SELECT query on the Student database table. The regenerated function will loop over each resulting row and add a tuple, which contains each value in the row, to the output list. The output list will then be returned at the end of the regenerated function.

NERO can also perform additional optimizations to simplify the regenerated program. For example, if a command returns a single value, NERO can detect this and return the value directly, instead of returning a list which contains a single value. Alternatively, in the example given above, instead of looping over each row returned by a SELECT query and adding it to a list, NERO can detect that the results of the SELECT query are the only values returned. NERO can therefore modify the regenerated function to return the results of the SELECT query directly. Finally, note that NERO outputs a regenerated program’s source code, which can be read and modified by a human. If the user expects the regenerated Python function to return the desired value in a slightly different format, they can perform the modification themselves, even if they do not understand the database queries performed by the regenerated program.

As discussed above, modifying the regeneration algorithm to produce a program which returns values is a straightforward change. However, this change does not add anything to the proof-of-concept provided by the prototype NERO implementation, because the proof-of-concept requires only that NERO can regenerate a program implementing the same functionality as the seed program but which uses a database instead of data structures. By regenerating a program which prints output to the console, the regenerated program more closely replicates the behavior of the seed program. Close replication makes it easier to automatically verify (as discussed in Section 8.4) that the seed program and regenerated program implement the same behavior. For these reasons, NERO as presented in this thesis regenerates a program which prints output to the console.


9.5 Regenerated Code Optimizations

As discussed in Section 7.1.2, the NERO regeneration algorithm performs optimizations on an intermediate representation of the regenerated program. These optimizations simplify the regenerated program and make it more efficient. Besides the two optimizations described in Section 7.1.2, I suggest the following optimizations:

- Unused context removal: The regenerated program uses a context list to track intermediate values. Many of the entries in the context are not accessed by any function call of the regenerated program.\(^4\) Unused context entries can be optimized out using standard compiler techniques for eliminating unused variables.

- Unused column removal: When the regenerated program regenerates a data structure access as a `SELECT` query, the regenerated query will retrieve every column in the database table. However, the program might not actually use the values from some of these columns. The regenerated program could be optimized to not retrieve values from unused columns. This optimization can be performed using standard compiler techniques for eliminating unused variables.

- Sequential access combination: Seed programs may contain a sequence of list index or dict key lookup operations. For example, the seed program might retrieve a value from a list, then use the value to index into another list. Currently, the regenerated program regenerates each lookup operation as an individual `SELECT` query. However, the lookups could be combined into a single `SELECT` query with a `JOIN` operation. This optimization can be performed in a similar manner to the two optimizations I described in Section 7.1.2.

These optimizations will result in a simpler and more efficient regenerated program. However, I did not include them in the prototype NERO implementation because each optimization is straightforward to implement and does not add anything to the proof-of-concept provided by the regeneration algorithm.

\(^4\)Usually, unused context entries are placeholders meant to ensure that context numbering remains consistent when function calls do not have return values.
9.6 Regenerated Code Readability

Currently, the regenerated code produced by the NERO inference algorithm is difficult to read. There are two main reasons for this:

- Intermediate values are stored in a context list, but humans are better at understanding programs which use variables to store intermediate values.

- Placeholders in SQL queries and print statements have unhelpful names, e.g. 
  
  ```
  ```{context_set_0} {context_set_1} {context_set_2}```

  from the student registration application example (Figure 2-8).

To improve readability of the regenerated code, the regeneration algorithm can use variables instead of a context list to store intermediate results. Because the length of the context list is known at regeneration-time, it is straightforward to replace each item in the context list with a variable, provided there is a way to generate the name of each variable. Because each variable stores data retrieved from a column of a database table, the variable name can be systematically generated based on the name of the column and database table. An index can be appended for uniqueness as needed. For example, a variable retrieved from a column named Name from a database table named Student might be `studentname`; and if another value is later retrieved from the same column, it might be named `studentname2`. The resulting variable names will not be as understandable as human-selected variable names, but because the user specifies the name of each table and column in the data specification file (Section 4.2), the systematically generated variable names should be comprehensible to a human. To improve readability of print statements and SQL queries, the name of each placeholder can be the same as the name of the variable containing the value for the placeholder.

As discussed above, it is relatively straightforward to implement these proposed changes to improve readability of the regenerated program. However, the proof-of-

---

5Nero generates format strings which use triple-quotes in case the printed string contains a newline.
6The exception to this is command parameters. The command specification file (Section 4.1) would need to additionally indicate the name of each command parameter.
7Assuming the human is following good coding practices.
concept provided by the prototype NERO implementation only requires that NERO is able to regenerate a program implementing the same functionality as the seed program. Therefore, NERO as presented in this thesis does not take these additional steps to enhance the readability of the regenerated program.

9.7 Different Regeneration Target

NERO regenerates a Python program which uses SQLite databases, to provide the simplest possible proof-of-concept that NERO can regenerate a program implementing the functionality of the seed program but using a database instead of data structures. However, a user may want to regenerate a program in a language other than Python, or to use a database other than a SQLite database. The purpose of NERO is to allow a user to take advantage of the benefits of a database without knowing how to implement a database program. NERO could similarly be extended to allow a user to take advantage of an efficient programming language like C++, or a powerful distributed database system, without a good understanding of either. NERO simply needs a new regeneration module which accepts as input the intermediate representation (IR) of the seed program’s functionality returned by the inference algorithm (Section 6.7).

Implementing a new and more flexible regeneration module is straightforward because the new regeneration module can be implemented in a very similar fashion to the regeneration algorithm I described for NERO in Chapter 7. The current NERO regeneration algorithm converts the IR returned by the inference algorithm to another IR representing the seed program functionality as a sequence of SQL queries (the SQL IR). The algorithm then optimizes the SQL IR and compiles it into a Python function accessing a SQLite database. A regeneration module which regenerates to a new language or different database type can use the same SQL IR and optimization algorithm; the only change would need to be the compilation step at the end. In other words, the only difference between two regeneration modules which generate to different languages or database types is the boilerplate generated. Therefore, it is straightforward, though potentially time-consuming, to implement a more flexible
9.8 Supporting More Query Operations

The NERO domain-specific language (DSL) allows seed programs to mimic the behavior of SELECT, UPDATE, INSERT INTO, and INSERT OR REPLACE INTO queries, potentially with WHERE or JOIN clauses (Section 7.1.2). NERO as presented in this paper supports seed programs which mimic these queries because these are the queries most naturally mimicked by Python data structure accesses. However, the NERO DSL could additionally be extended to allow seed programs to mimic the behavior of other queries as well:

- The IN clause: This clause is used in WHERE clauses, to indicate that a value should be a member of a particular set. By supporting IN clauses, NERO can expand the set of conditions seed programs are allowed to use. When looping over a data structure, a seed program could add values from a particular column to a temporary Python set data structure. After the set has been populated, the seed program could include a conditional loop, where the condition checks whether a value is a member of the set. When this functionality is inferred and regenerated, the resulting regenerated program will contain a SELECT query corresponding to the conditional loop. This SELECT query will contain a IN clause.

- The SELECT DISTINCT query: This query is a SELECT query that deduplicates values before returning them. A seed program currently imitates a SELECT query using a loop or conditional loop over a data structure.\(^8\) To imitate a SELECT DISTINCT query, the seed program could loop over a data structure and add each retrieved value to a Python set data structure, then iterate over the set and perform operations on it.

\(^8\)A list index or dict key lookup also mimics a SELECT query, but those queries return only a single row.
• The **ORDER BY** clause: This clause sorts the rows returned by a **SELECT** query. A seed program currently imitates a **SELECT** query using a loop or conditional loop over a data structure. To imitate a **ORDER BY** clause, the seed program could use the Python **sorted** function to sort the data structure by a particular column before performing the loop. The column could be specified using a Python lambda function, passed into the **sorted** function using the **key** keyword argument.

• The **LIMIT** clause: This clause limits the number of rows returned by a **SELECT** query. A seed program currently imitates a **SELECT** query using a loop or conditional loop over a data structure. To imitate a **ORDER BY** clause, the seed program could use a Python slicing operation to truncate the data structure before performing the loop.

• The **MIN**, **MAX**, **COUNT**, and **SUM** functions: When performing a SQL query, these functions can be used to summarize data in multiple rows at once. To imitate one of these functions, the seed program could loop (or conditionally loop) over a data structure and store values in a temporary list. Then, the seed program could call the built-in **min**, **max**, **len**, or **sum** Python functions on the temporary list.

To support the above functionality in seed programs, the **NERO** instrumentation would need to be modified to capture accesses to Python set data structures or to the built-in **sorted**,\(^\text{10}\) slicing, **min**, **max**, **len**, and **sum** operations. The inference algorithm would additionally need to be modified to correctly infer what happened when these function calls appear in the trace.

None of these proposed extensions to the **NERO** DSL require fundamental changes to the **NERO** inference algorithm, so they should be relatively easy to implement.

\(^9\)**AVG** is another common reducing function used in SQL, but it is not included in this list because it would introduce floating point numbers.

\(^{10}\)For **sorted**, the instrumentation would also perform some logic to determine what column is being sorted by. The instrumentation would be able to capture the lambda function used to specify the sort column, and could provide test inputs to the lambda function to determine which column is returned.
By adding this functionality, users would be able to use NERO to generate database programs which perform a wider variety of queries, increasing the number of practical applications for NERO.

9.9 Analytical Temperature Selection Algorithm

NERO uses a temperature variable to control the range of random values generated when executing the seed program, as described in Section 6.6.1. NERO controls the temperature for two purposes: to cause conditions to satisfy (Section 6.6.1) and to resolve ambiguity (Section 6.6.2). To cause conditions to satisfy, NERO begins with a moderate temperature, then slowly decreases the temperature until the condition is satisfied. To resolve ambiguity in a condition or function call input argument source, NERO rapidly decreases the temperature until the first time that the ambiguous condition satisfies or the first time the ambiguous function call is reached, then slowly raises the temperature. In both cases, NERO uses a simple algorithm to dynamically select the temperature based on the observed seed program behavior.

When selecting the temperature to cause conditions to satisfy or to resolve ambiguity, NERO has already inferred information about the seed program’s behavior, but this inferred information is not used in the temperature selection algorithm. For example, when causing a condition to satisfy, NERO has inferred every function call which occurs before the condition is reached. NERO can analyze the functionality which has already been inferred, and compute the optimal temperature to use when causing conditions to satisfy or when resolving ambiguity.

To compute the optimal temperature to use when causing conditions to satisfy or when resolving ambiguity, the calculations made in Section 6.8.5 should be a good starting point. In Section 6.8.5, I derived an estimate of the optimal temperature to use to resolve ambiguity in a function call input argument based on some properties of the seed program. These properties can be computed based on the functionality NERO infers, without needing to resolve the ambiguity first. However, more analysis needs to be done to determine the best temperature to use in every situation.
For reasons of simplicity and brittleness, NERO currently does not use an analytical algorithm which computes the optimal temperature. The algorithms currently used by NERO to select the temperature may not be optimal, but are simple to implement and understand, and produce “good-enough” behavior (as discussed in Section 6.8.5). In contrast, an analytical temperature selection algorithm would require a large amount of work to derive or understand the optimal temperature, and would likely require making certain assumptions about the seed program. If these assumptions are violated, for example if the NERO DSL is expanded to support a wider range of seed programs, it is possible an analytical temperature selection algorithm will systematically produce poor behavior.

Nevertheless, it is possible that putting work into developing an analytical temperature selection algorithm will result in a faster and more efficient inference algorithm. As I discussed in Section 6.8.5, the difficulty of resolving all ambiguity in an adversarial seed program can be exponential in the length of the program. By analytically computing the optimal temperature to use, NERO have better performance in these adversarial cases.

### 9.10 Analytical Input Solver

NERO currently uses a randomized constraint solver (Section 6.2) to determine the data and inputs for each execution of the seed program. The NERO inference algorithm adds and removes constraints and adjusts a temperature parameter (Section 6.6.1) to generate data and inputs which lead to different behavior in the seed program. Causing different behavior in the seed program allows NERO to progressively infer more functionality in the seed program until all of the functionality in the seed program has been inferred.

Instead of this randomized algorithm, one could use an analytical solver to model the inferred behavior of a seed program and to solve for a set of data and inputs which cause any arbitrary condition to satisfy, or which resolve an arbitrary ambiguity. This approach would allow NERO to deterministically infer the functionality of a seed
As I discussed in Section 6.4.1, there are advantages and disadvantages to using a solver in this manner. The main disadvantage is that it would take considerable time to design and implement an approach which uses an analytical solver to generate data and inputs for each execution of the seed program. Due to its complexity, such a solver may be prone to bugs. In contrast, the randomized solver currently used by Nero is easy to understand and easy to implement. However, using an analytical solver will allow Nero to deterministically and unambiguously infer the functionality of a seed program, even a complex one. In contrast, although the randomized approach is able to fully eliminate ambiguity with high probability in most practical seed programs, the randomized approach can take an exponentially long time (or have an exponentially low probability) of fully eliminating ambiguity in an adversarial seed program, as discussed in Section 6.8.5.
Chapter 10

Related Work

Program Inference and Regeneration The closest related work observes the data access patterns for a database-backed program, uses heuristic techniques to derive models of the program components that issue the database queries, and regenerates a new program with similar functionality [17]. In contrast to starting with an existing database-backed program, Nero starts with a Python seed program that uses built-in data structures, which is simpler to implement than a full-scale database application. Nero infers the functionality for this seed program and uses the inferred functionality to regenerate a new program for the database platform, which is more sophisticated and more efficient than the seed program. In contrast to using heuristic rules to infer functionality, Nero uses instrumentation to directly observe data structure accesses and is able to infer a more general class of programs.

Another prior work uses a black-box approach to infer functionality for a seed program and regenerate the inferred functionality on new platforms [16]. This technique works with seed programs that stores and retrieves data. In contrast to treating the seed program as a black box, Nero takes a gray box approach. It observes not only the inputs and outputs but also the execution trace in terms of function calls. As a result, Nero is able to support a more expressive domain-specific language than the prior work.

Object-Relational Mapping Object-relational mapping (ORM) frameworks [1, 6] allow developers to write object-oriented programs and transform objects into database
tables. In contrast, NERO does not require code analysis, which allows developers to use built-in Python syntax, which is widely taught, widely used, and relatively easy to learn.

**Optimizing Database-Backed Programs** Re-engineering database-backed programs often involves extracting data-access patterns from the source code and transforming these access patterns into more efficient database queries [8,9]. In contrast, NERO does not require code analysis, which allows the seed program to be written in flexible styles as long as the execution trace is compatible with NERO’s model.

**Program Synthesis** Synthesis-by-example techniques often require user-specified input-output pairs as the specification [7,10,11,13,15,18]. In contrast, NERO automatically generates interesting inputs and data structure contents.

Other synthesis techniques often involve solving constraints [12,14], applying templates [19–21], or using hypothetical I/O oracles [13] to solve for programs that satisfy the specification. These techniques do not work with existing implementations, but require abstract specifications in other forms. NERO, in contrast, works with an existing seed program.
Chapter 11

Conclusion

In this thesis, I presented new active learning techniques which automatically infer an abstract representation of the functionality implemented by a seed program written in Python. Below, I discuss the key aspects of my approach as well as the implications of the techniques presented in this thesis.

11.1 **NERO Key Aspects**

Key aspects of the techniques presented in this thesis include:

1. The formulation of an inferrable subset of Python that supports a range of computational patterns in database-backed applications (Chapter 3).

2. Instrumented data structures that make it possible to observe the sequence of database operations (Chapter 5).

3. The inference algorithm, which progressively synthesizes inputs and data structure contents that productively resolve uncertainty in the current inference result (Chapter 6). The inference algorithm is designed to infer the functionality of seed programs which do not contain any input validation code, relieving the user of the burden of writing input validation code in the seed program. As I argued in Section 6.8, the inference algorithm will always infer the correct structure of the functionality of the seed program, and will never eliminate the
true functionality of the seed program as a possibility; and the algorithm will, for seed programs encountered in practice, eliminate all uncertainty with high probability.

4. The regeneration algorithm, which encapsulates the knowledge of how to use databases produce augmented programs that contain systematically generated interfaces and database queries that implement the desired core functionality (Chapter 7). The regeneration algorithm also automatically generates input validation checks.

Results from my implementation highlight the ability of this approach to infer my four benchmark seed programs and regenerate database-backed implementations for them. Below, I discuss the implications of this thesis.

11.2 Implications

The presented techniques in this thesis hold out the promise of improving the productivity of database programmers and enable developers with little database expertise to obtain programs that access databases. This increase in productivity will increase the accessibility of database programs and decrease the cost of developing database-backed software. Databases are used to store information in a wide array of fields in science in technology. Therefore, the cost of performing research or engineering promises to decrease in a wide range of fields. Decreasing the cost of research or engineering could lead to more rapid advances in every area of society.

More generally, the inference techniques presented in this thesis dynamically explore the functionality of a program where each of the intermediate steps in the program can be observed. By using a dynamic algorithm to explore the functionality of the program, this approach is able to abstract away the program’s implementation details and automatically extract a program’s core functionality. This approach might be used to infer the functionality of an antiquated or inefficiently-written program and regenerate a modern or more efficient program to replace it. This approach might
be used to decompile executable files by inferring the functionality of the executable and regenerating source code. The resulting source code might then be improved or repurposed. The inference approach presented in this thesis might be used to automatically ask questions of a domain expert and learn how the domain expert makes decisions. In each of these cases, the inference techniques presented in this thesis promise to improve the quality of software available to society or increase the breadth of what software can accomplish.

The techniques presented in this thesis are part of a new approach to automated programming. The current state-of-the-art program synthesis techniques often use user-specified input-output pairs to specify the target program [7, 10, 11, 13, 15, 18], leading to a large search space of candidate programs, which generalizes poorly to synthesizing large or complex programs. In contrast, the techniques presented in this thesis focus on a particular domain of programs, namely programs which access databases to store data. The techniques in this thesis use an easy-to-write seed program to specify the target program. By focusing on a particular domain, the techniques presented in this thesis allow the core functionality of the seed program to be extracted by observing execution traces. The inferred functionality can then be regenerated into the desired target program, which may exhibit desired properties (in the case of Nero, usage of a database) which the seed program does not. Nero has made a successful step towards the synthesis of complex programs using this technique. This success may inspire new ways to define the problem of automated programming.

Overall, this thesis is a step forward in the automation of programming. The automation of programming will increase availability and decrease the cost of software. It will also allow increasing numbers of people to write software for their own uses. Software development for database-backed programs will no longer be a field accessible only to the highly skilled, but will instead be something available to all.
Appendix A

Code

In this appendix, I include the seed program code and regenerated program code for each of the four potential applications discussed in Chapter 8.
card_comments = [] # id -> content, card_id
cards = [] # id -> title, list_id
lists = [] # id -> board_id
boards = [] # id -> name
board_members = [] # board_id, member_id
users = [] # id -> name, email, password

def get_api_board(email, password):
    for user_id, (name, user_email, user_password) in enumerate(users):
        if not (user_email == email and user_password == password):
            continue
        for board_id, member_id in board_members:
            if member_id != user_id:
                continue
            board_name = boards[board_id]
        for list_id, list_board_id in enumerate(lists):
            if list_board_id != board_id:
                continue
        for card_id, (card_title, card_list_id) in enumerate(cards):
            if card_list_id != list_id:
                continue
            for comment_id, (content, comment_card_id) in enumerate(card_comments):
                if comment_card_id != card_id:
                    continue
            print(
                "user_id {}, user_name {}, user_email {}, "
                "board_id {}, board_name {}, "
                "list_id {}, "
                "card_id {}, card_title {} "
                "comment_id {}, comment_content {}"
                .format(
                    user_id, name, email,
                    board_id, board_name,
                    list_id,
                    card_id, card_title,
                    comment_id, content))

Figure A-1: Seed program for the task management application. This command checks user identity, checks user privilege, and then retrieves all cards visible to the user.
def do_get_api_board(args):
    context = [(a,) for a in args]
    context.append(None)
    query = 'SELECT Users.id - 1, Users.name, Users.email, Users.password, BoardMembers.id - 1, BoardMembers.board_id, BoardMembers.member_id, Boards.id - 1, Boards.name FROM Users INNER JOIN BoardMembers ON BoardMembers.member_id = Users.id - 1 INNER JOIN Boards ON Boards.id - 1 = BoardMembers.board_id WHERE Users.email = :lookup0 AND Users.password = :lookup1'
    params = {'lookup1': context[1][0], 'lookup0': context[0][0]}
    context.append(c.execute(query, params).fetchall())
    for i0, _ in enumerate(context[3]):
        context.append(None)
        context.append((context[3][i0][4], context[3][i0][5], context[3][i0][6], context[3][i0][7], context[3][i0][8]))
        context.append(context[5][4], )
        context.append(None)
        query = 'SELECT Lists.id - 1, Lists.board_id, Cards.id - 1, Cards.title, Cards.list_id, Comments.id - 1, Comments.content, Comments.card_id FROM Lists INNER JOIN Cards ON Cards.list_id = Lists.id - 1 INNER JOIN Comments ON Comments.card_id = Cards.id - 1 WHERE Lists.board_id = :lookup'
        params = {'lookup': context[5][1]}
        context.append(c.execute(query, params).fetchall())
        for i1, _ in enumerate(context[8]):
            context.append(None)
            context.append((context[8][i1][2], context[8][i1][3], context[8][i1][4], context[8][i1][5], context[8][i1][6], context[8][i1][7], context[8][i1][8]))
            context.append(None)
            context.append((context[10][3], context[10][4], context[10][5]))
        print('''user_id {context_set_0}, user_name {context_set_1}, user_email {context_set_2}, board_id {context_set_3}, board_name {context_set_4}, list_id {context_set_5}, card_id {context_set_6}, card_title {context_set_7}'''.format(**context))
    context.append(None)
    context = context[:-5]
    context = context[:-1]
    context = context[:-4]
    context = context[:-1]

Figure A-2: Regenerated program for the task management application. This command checks user identity, checks user privilege, and then retrieves all cards visible to the user.
def get_api_cards_id(email, password, card_id):
    card_id = int(card_id)
    for user_id, (name, user_email, user_password) in enumerate(users):
        if user_email != email or user_password != password:
            continue
    card_title, list_id = cards[card_id]
    board_id = lists[list_id]
    board_name = boards[board_id]
    # check board membership
    for member_board_id, member_id in board_members:
        if member_id == user_id and member_board_id == board_id:
            for comment_id, (content, comment_card_id) in enumerate(card_comments):
                if comment_card_id != card_id:
                    continue
                print(
                    "user_id {}, user_name {}, user_email {}, "
                    "board_id {}, board_name {}, "
                    "list_id {}, "
                    "card_id {}, card_title {} "
                    "comment_id {}, comment_content {}"
                .format(
                    user_id, name, email,
                    board_id, board_name,
                    list_id,
                    card_id, card_title,
                    comment_id, content))

Figure A-3: Seed program for task management application (continued). This command checks user identity, checks user privilege, and then retrieves a card specified by the input card ID.
def do_get_api_cards_id(args):
    context = [(a,) for a in args]
    context.append(None)
    query = 'SELECT Users.id - 1, Users.name, Users.email, Users.password FROM Users WHERE Users.email = :lookup0 AND Users.password = :lookup1'
    params = {'lookup1': context[1][0], 'lookup0': context[0][0]}
    context.append(c.execute(query, params).fetchall())
    for i0, _ in enumerate(context[4]):
        query = 'SELECT Cards.id - 1, Cards.title, Cards.list_id FROM Cards WHERE Cards.id - 1 = :index'
        params = {'index': context[2][0]}
        res = c.execute(query, params).fetchone()
        if res is None:
            print('No result found.')
            return
        context.append(res[1:])
    context.append(res[1:])
    query = 'SELECT Lists.id - 1, Lists.board_id FROM Lists WHERE Lists.id - 1 = :index'
    params = {'index': context[5][0]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found.')
        return
    context.append(res[1:])
    query = 'SELECT Boards.id - 1, Boards.name FROM Boards WHERE Boards.id - 1 = :index'
    params = {'index': context[6][0]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found.')
        return
    context.append(res[1:])
    query = 'SELECT BoardMembers.idx - 1, BoardMembers.board_id, BoardMembers.member_id FROM BoardMembers
WHERE BoardMembers.board_id = :lookup0 AND BoardMembers.member_id = :lookup1'
    params = {'lookup1': context[4][i0][0], 'lookup0': context[6][0]}
    context.append(c.execute(query, params).fetchall())
    for i1, _ in enumerate(context[9]):
        context.append(None)
    query = 'SELECT Comments.id - 1, Comments.content, Comments.card_id FROM Comments WHERE Comments.card_id = :lookup'
    params = {'lookup': context[2][0]}
    context.append(c.execute(query, params).fetchall())
    for i2, _ in enumerate(context[11]):
        context.append(None)
    context = context[:-1]
    context = context[:-1]
    context = context[:-1]
    context = context[:-4]
    context = context[:-1]
    context = context[:-1]

Figure A-4: Regenerated program for the task management application (continued).
This command checks user identity, checks user privilege, and then retrieves a card
specified by the input card ID.
def post_api_lists(email, password, board_id):
    board_id = int(board_id)
    for user_id, (_, user_email, user_password) in enumerate(users):
        if not (user_email == email and user_password == password):
            continue
        for member_board_id, member_id in board_members:
            if member_id == user_id and member_board_id == board_id:
                lists.append(board_id)

def post_api_cards(email, password, title, list_id):
    list_id = int(list_id)
    board_id = lists[list_id]
    for user_id, (_, user_email, user_password) in enumerate(users):
        if not (user_email == email and user_password == password):
            continue
        for member_board_id, member_id in board_members:
            if member_id == user_id and member_board_id == board_id:
                cards.append((title, list_id))

Figure A-5: Seed program for the task management application (continued). The first command checks user identity, checks user privilege, and then creates a new list in the specified board. The second command checks user identity, checks user privilege, and then creates a new card in the specified list.
Figure A-6: Regenerated program for the task management application (continued). The first command checks user identity, checks user privilege, and then creates a new list in the specified board. The second command checks user identity, checks user privilege, and then creates a new card in the specified list.
users = []  # id -> username, email, password, first_name, last_name
activities = []  # id -> user_id, channel_id, content
channels = []  # id -> name, user_id

def do_get_channels(username_query, password_query):
    for user_id, (username, email, password, first_name, last_name) in enumerate(users):
        if not (username == username_query and password == password_query):
            continue
        print(user_id, username, email, first_name, last_name)
        for channel_id, (channel_name, channel_user_id) in enumerate(channels):
            print(channel_id, channel_name, channel_user_id)
            for activity_id, (activity_user_id, activity_channel_id, content) in enumerate(activities):
                if channel_id != activity_channel_id:
                    continue
                print(activity_id, content)
                for user_id2, (username2, email2, first_name2, last_name2) in enumerate(users):
                    if activity_user_id != user_id2:
                        continue
                    print(user_id2, username2, email2, first_name2, last_name2)

def do_post(username_query, password_query, content, channel_id_query):
    for user_id, (username, password, _, _, _) in enumerate(users):
        if username == username_query and password == password_query:
            for channel_id, _ in enumerate(channels):
                if channel_id == channel_id_query:
                    activities.append((user_id, channel_id, content))

Figure A-7: Seed program for the chat room application. The first command checks
user identity and then prints all chat activities as well as their authors. The second
command checks user identity and then creates a new chat activity with the specified
content in the specified channel.
```python
def do_get_channels(args):
context = [(a,) for a in args]
context.append(None)
query = 'SELECT Users.user_id - 1, Users.username, Users.email, Users.password
    , Users.firstname, Users.lastname FROM Users WHERE Users.username = :lookup0
    AND Users.password = :lookup1'
params = {'lookup1': context[1][0], 'lookup0': context[0][0]}
context.append(c.execute(query, params).fetchall())
for i0, _ in enumerate(context[3]):
    print(f'\n\n{context[i0][1]} {context[i0][2]} {context[i0][3]} {context[i0][4]} {context[i0][0]}\n\n')
context.append(None)
context.append(None)
query = 'SELECT Channels.channel_id - 1, Channels.name, Channels.user_id
    FROM Channels'
params = {}  
context.append(c.execute(query, params).fetchall())
for i1, _ in enumerate(context[6]):
    print(f'\n\n{context[i1][1]} {context[i1][2]} {context[i1][0]}\n\n')
context.append(None)
context.append(None)
query = 'SELECT Activities.activity_id - 1, Activities.user_id, Activities
    .channel_id, Activities.content FROM Activities WHERE Activities.channel_id = :lookup'
params = {'lookup': context[9][i2][0]}
context.append(c.execute(query, params).fetchall())
for i2, _ in enumerate(context[9]):
    print(f'\n\n{context[i2][1]} {context[i2][0]}\n\n')
context.append(None)
context.append(None)
query = 'SELECT Users.user_id - 1, Users.username, Users.email, Users.
    password, Users.firstname, Users.lastname FROM Users WHERE Users.user_id - 1
    = :lookup'
params = {'lookup': context[9][i2][1]}
context.append(c.execute(query, params).fetchall())
for i3, _ in enumerate(context[12]):
    print(f'\n\n{context[i3][1]} {context[i3][2]} {context[i3][3]} {context[i3][4]} {context[i3][5]} {context[i3][6]} {context[i3][7]} {context[i3][8]} {context[i3][9]} {context[i3][0]}\n\n')
context.append(None)
context = context[:-1]
context = context[:-2]
context = context[:-1]
context = context[:-2]
context = context[:-1]
context = context[:-2]
context = context[:-1]
context = context[:-2]
context = context[:-1]
```

Figure A-8: Regenerated program for the chat room application. This command checks user identity and then prints all chat activities as well as their authors.
def do_post(args):
    context = [(a,) for a in args]
    context.append(None)
    query = 'SELECT Users.user_id - 1, Users.username, Users.email,'
            ' Users.password, Users.firstname, Users.lastname FROM Users
            WHERE Users.username = :lookup0 AND Users.email = :lookup1'
    params = {'lookup1': context[1][0], 'lookup0': context[0][0]}
    context.append(c.execute(query, params).fetchall())
    for i0, _ in enumerate(context[5]):
        context.append(None)
        query = 'SELECT Channels.channel_id - 1, Channels.name,'
                 ' Channels.user_id FROM Channels WHERE Channels.channel_id - 1
             = :lookup'
        params = {'lookup': context[3][0]}
        context.append(c.execute(query, params).fetchall())
        for i1, _ in enumerate(context[7]):
            query = 'INSERT INTO Activities (user_id, channel_id,
            content) VALUES (:user_id, :channel_id, :content)'
            params = {'user_id': context[5][i0][0], 'channel_id':
                    context[3][0], 'content': context[2][0]}
            context.append(c.execute(query, params))
    context = context[:-1]
    context = context[:-1]
    context = context[:-1]

Figure A-9: Regenerated program for the chat room application (continued). This
command checks user identity and then creates a new chat activity with the specified
content in the specified channel.
1 articles_list = []  # id: (title, text)
2 comments = []  # id: (article_id, commenter, body)
3
4 def articles():
5     for i, (title, text) in enumerate(articles_list):
6         print("{0} {1}: {}").format(i, title, text))
7
8 def article(article_id):
9     title, text = articles_list[article_id]
10    print("{0} {1}: {}").format(article_id, title, text))
11   for a_id, commenter, body in comments:
12       if a_id == article_id:
13           print("{0} {1}: {}").format(commenter, body))
14
15 def new_article(arg_title, arg_text):
16    articles_list.append((arg_title, arg_text))
17
18 def new_article_comment(article_id, commenter, body):
19    comments.append((article_id, commenter, body))

Figure A-10: Seed program for the blog application. The first command prints all articles. The second command prints the specified article, along with all comments for this article. The third command creates a new article. The last command creates a new comment for an article.
def do_articles(args):
    context = [(a, ) for a in args]
    context.append(None)
    query = 'SELECT Articles.article_id - 1, Articles.title, Articles.text FROM Articles
    params = {} 
    context.append(c.execute(query, params).fetchall())
    for i0, _ in enumerate(context[1]):
        print('''({context_set_0}) {context_set_1}: {context_set_2}'''.format(
            context_set_2=context[1][i0][2], context_set_1=context[1][i0][1],
            context_set_0=context[1][i0][0]))
    context.append(None)
    context = context[:-1]

    context = context[:-1]

    def do_article(args):
        context = [(a, ) for a in args]
        query = 'SELECT Articles.article_id - 1, Articles.title, Articles.text FROM Articles WHERE Articles.article_id - 1 = :index'
        params = {'index': context[0][0]}
        res = c.execute(query, params).fetchone()
        if res is None:
            print('No result found. ')
            return
        context.append(res[1:])
        print('''({context_set_0}) {context_set_1}: {context_set_2}'''.format(
            context_set_2=context[1][1], context_set_1=context[1][0], context_set_0=context[0][0]))
        context.append(None)
        context.append(None)

    def do_new_article(args):
        context = [(a, ) for a in args]
        query = 'INSERT INTO Articles (title, text) VALUES (:title, :text)' 
        params = {'title': context[0][0], 'text': context[1][0]} 
        context.append(c.execute(query, params))

    def do_new_article_comment(args):
        context = [(a, ) for a in args]
        query = 'INSERT INTO Comments (article_id, commenter, body) VALUES (:article_id, :commenter, :body)'
        params = {'article_id': context[0][0], 'commenter': context[1][0], 'body': context[2][0]}
        context.append(c.execute(query, params))

Figure A-11: Regenerated program for the blog application. The first command prints all articles. The second command prints the specified article, along with all comments for this article. The third command creates a new article. The last command creates a new comment for an article.
def do_list_student_courses(args):
    context = [(a,) for a in args]
    query = 'SELECT Student.id - 1, Student.first_name, Student.last_name FROM
    Student WHERE Student.id - 1 = :index'
    params = {'index': context[0][0]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found. ')
        return
    context.append(res[1:])
    print('''Student: {context_set_0}, {context_set_1}'''.format(context_set_1=context[1][1],
        context_set_0=context[1][0]))
    context.append(None)
    context.append(None)
    query = 'SELECT Registration.reg_id - 1, Registration.student_id, Registration.
course_num, Course.course_num, Course.name, Course.teacher_id FROM
    Registration INNER JOIN Course ON Course.course_num = Registration.
course_num WHERE Registration.student_id = :lookup'
    params = {'lookup': context[0][0]}
    for i0, _ in enumerate(context[4]):
        context.append(c.execute(query, params).fetchall())
        context.append(None)
        context.append((context[4][i0][3], context[4][i0][4], context[4][i0][5]))
    query = 'SELECT Course.course_num, Course.name, Course.teacher_id FROM
    Course WHERE Course.course_num = :key'
    params = {'key': context[4][i0][2]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found. ')
        return
    context.append(res[1:])
    print('''{context_set_0} {context_set_1} {context_set_2}'''.format(
        context_set_2=context[7][1], context_set_1=context[7][0],
        context_set_0=context[4][i0][2]))
    context.append(None)
    context = context[:-4]

def do_register(args):
    context = [(a,) for a in args]
    query = 'SELECT Student.id - 1, Student.first_name, Student.last_name FROM
    Student WHERE Student.id - 1 = :index'
    params = {'index': context[0][0]}
    res = c.execute(query, params).fetchone()
    if res is None:
        print('No result found. ')
        return
    context.append(res[1:])
    print('''{context_set_0} {context_set_1}''''.format(context_set_1=context
        [2][1], context_set_0=context[2][0]))
    context.append(None)
    query = 'INSERT INTO Registration (student_id, course_num) VALUES (:student_id
    , :course_num)'
    params = {'course_num': context[1][0], 'student_id': context[0][0]}
    context.append(c.execute(query, params))

Figure A-12: Regenerated program for the student registration application. The first
command prints the student and then lists all courses for the student. The second
command adds the specified student to the specified course.
Bibliography


