Dynamic and Interactive Synthesis of Code Snippets

Joel Galenson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-160
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-160.html

August 20, 2014

Dynamic and Interactive Synthesis of Code Snippets

by

Joel David Galenson

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodík, Co-chair
Professor Koushik Sen, Co-chair
Professor Björn Hartmann
Professor Leo A. Harrington

Fall 2014

Dynamic and Interactive Synthesis of Code Snippets

Abstract

Dynamic and Interactive Synthesis of Code Snippets

by

Joel David Galenson

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodík, Co-chair

Professor Koushik Sen, Co-chair

Many code fragments are difficult to write. For example, using new and unfamiliar APIs can be a complex task with a steep learning curve. In addition, implementing a complex data structure requires discovering and understanding all of the corner cases. And more and more end users with little to no formal training are trying to write code, whether they be scientists writing simulations or kids writing mobile apps. For all of these reasons and more, programming is a difficult task, which leads to bugs and delays in software.

There are many tools that help programmers find code fragments involving complex APIs, but many are somewhat inexpressive and rely on static information. We present a new technique, which we call `CodeHint`, that generates and evaluates code at runtime and hence can synthesize real-world Java code that involves I/O, reflection, native calls, and other advanced language features. Our approach is dynamic (giving accurate results and allowing programmers to reason about concrete executions), easy-to-use (supporting a wide range of correctness specifications), and interactive (allowing users to refine the candidate code snippets). We evaluate `CodeHint` and show that its algorithms are efficient and that in two user studies it improves programmer productivity by more than a factor of two.

As the second contribution, programmers and end users often find it easy to explain an algorithm on a whiteboard or with pictures in a textbook but struggle to write the code correctly. We propose a new methodology that allows users to program by demonstrating how an algorithm proceeds on concrete inputs. To reduce the burden of these demonstrations on the user, we have developed pruning algorithms to remove ambiguities in the demonstrations and control flow inference algorithms to infer missing conditionals in demonstrations. These two techniques take advantage of the knowledge encoded in the user's partial correctness condition. We show that this approach is effective in practice by analyzing its performance on several common algorithms.

To my family.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I would like to thank my advisors, Rastislav Bodík and Koushik Sen, for the ideas, feedback, and time they have given me over the past few years.

I would also like to thank Björn Hartmann and Leo Harrington for serving on my qualifying exam and dissertation committees and for giving me useful feedback and helping to shape my thesis.

My collaborators, including Philip Reames, Mihai Budiu, Gordon D. Plotkin, David Gay, Mayur Naik, Satish Chandra, and Vijay Saraswat, were instrumental in helping me discover interesting problems and design solutions to them, for which I thank them.

I am very grateful to the amazing undergradutes, graduate students, and postdocs I have worked with here at Berkeley, including Gilad Arnold, Eric Atkinson, Shaon Barman, Jacob Burnim, Sarah Chasins, Wontae Choi, Liang Gong, Thibaud Hottelier, Nick Jalbert, Pallavi Joshi, Ali Sinan Köksal, Patrick Li, Leo Meyerovich, Cuong Nguyen, Chang-Seo Park, Mangpo Phothilimthana, Michael Pradel, Evan Pu, Xuehai Qian, Cindy Rubio González, Rohit Sinha, Saurabh Srivastava, Christos Stergiou, Emina Torlak, Matt Torok, Nishant Totla, Nicholas Tung, and Chenyang Yuan. Their feedback on papers, talks, and ideas as well as general discussions were invaluable. I would also like to thank Roxana Infante, Tamille Johnson, and Lydia Raya for their organizational help.

Finally, I would like to thank my family for supporting me all these years.

Chapter 1

Introduction

The process of writing computer programs is difficult and developers spend a significant amount of their time trying to write correct code. As software becomes more prevalent and programs become larger, this cost will only increase.

Furthermore, many bugs still remain in final products. The Ariane 501 rocket crashed due to a software overflow bug [57] and the Therac-25 gave several people severe overdoses of radiation due to a race condition [53]. These and other major incidents have had significant costs to the world economy [100, 87].

As programs become larger and higher-level languages become more popular, more programmers are writing code involving unfamiliar APIs. Unfortunately, such code can be difficult to write due to poor documentation [42]. As an example, the Mars Climate Orbiter crashed due to an inconsistency between two APIs [83]. Helping programmers write correct code using complex APIs could thus greatly improve programmer productivity and reduce bugs.

In addition, the increase in the importance of computer software has led to more people trying to write software [71]. Helping teach students or even end-users to program has thus become more important.

In this thesis we develop techniques for helping these two types of programmers based on program synthesis [61], which aims to write correct code snippets for programmers. Our `CodeHint` technique uses the user's specification to synthesize code involving complex APIs, while our work on Programming by Demonstration makes it easier for students, end-users, and some developers to program without actually writing code.

1.1 CodeHint

Many code fragments are difficult to write, often because they involve using a new and unfamiliar API. Programmers have many tools at their disposal, from search and autocomplete to advanced synthesis techniques, but these often have only limited applicability. For example, autocomplete implementations can be helpful at finding a method to call, but they require

some knowledge of the static type of the receiver and can only generate tiny code fragments. Previous approaches based on API exploration [59, 44, 74, 88] can find larger code fragments but usually rely on simple specifications and some static analysis of the codebase. Recently, program synthesis [81, 48, 47] has been used to generate large code fragments, but these techniques often work only for certain domains and cannot handle arbitrary complex APIs.

To make program synthesis techniques easier to use, we believe that programmers need to be able to give partial specifications (rather than full correctness conditions) that may depend on concrete program state. These specifications, which can be any predicate in the host language, encode high-level hints about the code that should be synthesized.

To allow our program synthesis approach to generate complex code, our algorithms generate and evaluate code snippets at runtime to see if they meet users' specifications. This allows us to synthesize complex code without modeling the entire language.

Users of our technique can execute their programs on a specific concrete input and choose a location to synthesize code. Our tool then generates potential code fragments and shows the users those that meet their specification.

We thus propose a new approach for synthesizing code snippets that is dynamic, easy-to-use, and interactive. Running in the dynamic context both allows us to find and filter out candidates that static techniques could not and allows users to reason concretely about their desired result. We support a wide variety of specifications so that we can even aid programmers with very little knowledge of their desired code. Our methodology is interactive, letting users incrementally give more information to refine the candidate code fragments.

Taking advantage of dynamic information allows our algorithms to be more accurate than static techniques by, for example, dereferencing exactly the expressions that do not evaluate to `null` in the current context and downcasting the result of a method call to its dynamic type to enable subsequent calls. In addition, users can use dynamic values in their specifications and see the results of executing the candidates, which can be helpful in choosing the correct result.

One major goal of our work has been to ensure that users can find code snippets using whatever partial information they have about the desired code simply by writing a predicate in the host language. These partial dynamic specifications or *pdspecs* give us the flexibility to represent a large spectrum of specification strength and context sensitivity. For example, our pdspecs can take the form of constraints on the desired value (including demonstrating the concrete value that is desired in a specific context), dynamic type restrictions (which might hold for all contexts), or even full functional correctness specifications. We also allow users to write code *skeletons* to shape the search space by giving a syntactic outline of the desired code with holes marking unknown fragments.

Given a set of candidate statements synthesized by our tool, users can refine this set by continuing to run the program or by exercising it on different inputs in order to filter out more candidates that fail the specification. This refinement process can quickly remove many undesirable code snippets. Users can also sort and filter the candidates and their results. These features often allow users to find their desired code even with simple pdspecs.

Another goal of our work is to enable the synthesis of code in real-world Java programs. Because it actually evaluates candidates in a concrete program state, our implementation can synthesize code that uses I/O, reflection, native calls, and more. We propose novel techniques for using standard features of JVMs such as breakpoints to ensure that our evaluations have no undesirable side effects, which users can selectively enable or disable to control the tradeoff between efficiency, soundness, and completeness. By analyzing over ten million lines of code, we have developed a probabilistic model of real-world Java code that helps guide our search toward more common methods and fields.

To demonstrate the benefits of our approach, we conducted two user studies involving 28 subjects solving multiple programming problems in different domains such as GUIs, string parsing, and Eclipse plugins. The statistically significant results show that subjects using our tool complete more tasks in less time and with fewer bugs than those without it by more than a factor of two. Users gave our tool positive subjective ratings.

1.2 Programming by Demonstration with Little User Interaction

The best way to explain an algorithm is often to demonstrate how it works. Many textbooks give graphical examples of algorithms operating on concrete inputs before formally defining them. Since these demonstrations are a useful way of teaching and explaining, we believe that they can be a good way to program. We thus propose allowing users to program by demonstrating their algorithms on concrete inputs rather than requiring them to program symbolically.

Previous work on programming by demonstration (PBD) has taken advantage of these insights by synthesizing a program from user-provided demonstrations [39, 24, 55]. Unfortunately, when generalizing the demonstrations, all PBD systems must deal with ambiguities (e.g., multiple expressions that evaluate to the demonstrated value). Some require multiple demonstrations of the same statement [50] while others rank the potential solutions to try to guess the correct one [35].

For example, consider trying to write code for a self-balancing binary search tree such as a red-black tree. This code can be difficult to write correctly, but the key insights can easily be described by demonstrating how the algorithm proceeds on a few examples, e.g., by manually manipulating pointers on a specific concrete input. However, the values used in these demonstrations can often be referred to by multiple names, such as a node that is both `tree` and `x.parent`. From a single demonstration that might not even include any names, a PBD system can thus not always generalize the code.

Our work is a combination of PBD, synthesis, and graphical programming techniques. Its core contribution is a methodology that improves on the ease of programming by demonstration with new techniques to resolve the ambiguities and missing branches in the demonstrations. We do this with a combination of algorithmic approaches and interactively asking

the user for further demonstrations, where we try to require as little work from the user as possible.

Our approach works by initially asking the user to demonstrate the algorithm on a sample concrete input. Our tool then automatically generates new inputs and uses them to resolve ambiguities in this demonstrated trace as well as to find failing inputs. These failing inputs are often due to conditional branches that were not seen in the initial trace. Our tool then walks the user through the program on one of these new failing inputs to learn more about the code. It continues this process until it has synthesized as much of the code as it can.

To automatically resolve ambiguities without overly burdening the user, our approach can take advantage of user-provided tests and partial specifications. After each new demonstration, our tool will re-run existing tests and generate new ones so that it can ignore potential code that fails a test or the correctness condition. This process can greatly reduce the burden on the user.

Returning to the red-black tree example, we can automatically resolve some ambiguities by discovering that certain candidate expressions will crash or lead to invalid outputs on inputs we have not seen before. When we are unable to completely resolve an ambiguity, we can generate a new input that will disambiguate it, such as one where `tree` and `x.parent` are different, and ask the user to demonstrate a trace on this input. We can also automatically infer that users have likely not demonstrated all of the branches their algorithms contain by finding edge cases that cause them to crash, at which point we will generate an input we suspect has such a branch and ask the users to demonstrate their algorithms on it.

Our work closely integrates edge case discovery into the development process, as the missing conditional branches we discover often correspond to edge cases that can be difficult for users to recognize. In addition, our users need only demonstrate their algorithms on individual concrete inputs instead of having to consider abstract states or the interplay between different paths through the program.

While our approach should work with any imperative language, it fits naturally with graphical programming languages [71, 93, 69], which allow programmers to manipulate their programs graphically rather than textually. Algorithms that manipulate data structures such as lists and trees are an especially good fit, and novice programmers often find such graphical components appealing [71, 80]. However, our methodology is general and can in theory be used to encode any program.

To evaluate how effectively our algorithms can learn code while asking the user as few questions as possible, we present several case studies on common programming algorithms such as inserting an element in a red-black tree [34] and the Deutsch-Schorr-Waite stackless graph marking algorithm [75]. These studies show that our algorithms greatly reduce the number of queries we ask users, and in fact that we require only slightly more than the minimum amount of work, which suggests that our techniques improve the usability of PBD systems.

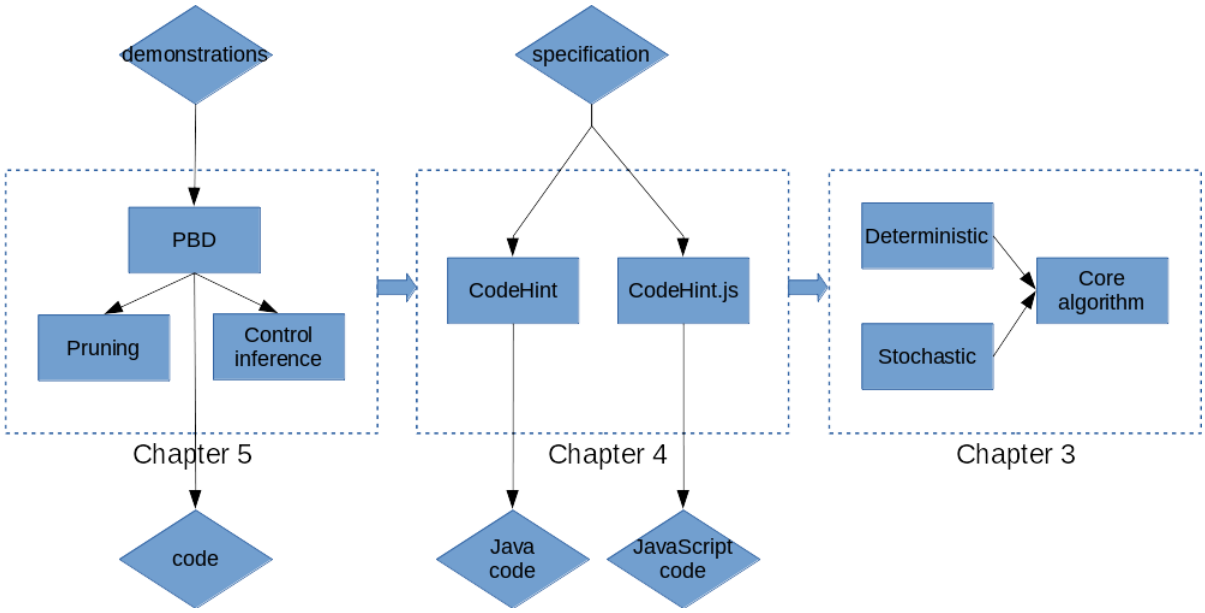


Figure 1.1: A high-level overview of our approach.

1.3 Contributions and Outline

We begin in Chapter 2 by giving a high-level overview of our techniques by walking through how they work on specific examples. As shown in Figure 1.1, Chapters 3, 4, and 5 then present the main technical content of our synthesis approaches and algorithms, including the following main contributions:

- A new approach (Section 3.1) for synthesizing code that is dynamic (allowing it to give accurate results and allowing programmers to reason about concrete executions), easy-to-use (supporting a wide range of correctness specifications), and interactive (allowing users to refine the candidate code snippets).
- An efficient algorithm (Section 3.2) that exploits the dynamic context to generate candidate statements that can include advanced features of the host language such as I/O, reflection, and native calls.
- An open-source implementation (Section 4.1) as a plugin for the Eclipse IDE called `CodeHint` that synthesizes Java code, including Android programs.
- An exploration (Section 4.2) of different probabilistic models for Java code and an analysis of their effectiveness at predicting methods that are called.
- Empirical evaluations and user studies (Section 4.3) that show that `CodeHint` is efficient and significantly improves programmer productivity.

- An open-source prototype implementation (Section 4.4) for JavaScript called `CodeHint.js` that synthesizes simple code snippets.
- A new programming by demonstration methodology (Section 5.1) that leverages our approach to allow users demonstrate their algorithms on concrete inputs and interactively asking for further demonstrations to determine the desired code.
- New algorithms (Section 5.2) for automatically resolving ambiguities in users' demonstrations, thereby reducing the number of questions we ask users.
- New techniques (Section 5.3) for inferring and completing missing conditionals in traces with little user interaction, which allows us to infer edge cases.
- An open-source prototype of our approach (Section 5.4) for a Java-like language and show that it can successfully synthesize code for a number of algorithms found in algorithms textbooks and that it greatly reduces the number of queries we ask the user (Section 5.5).

We then present related work in Chapter 6 and discuss future work and conclude in Chapter 7.

Chapter 2

Overview

We now give a high-level overview of our two core techniques. We begin in Section 2.1 by walking through two examples of how programmers can use `CodeHint` to help them write code involving new and unfamiliar APIs. We then walk though how users can use our Programming by Demonstration methodology to program a complex data structure in Section 2.2.

2.1 CodeHint

We now present two example problems and show how `CodeHint` can solve them. The first shows the user's perspective and the second demonstrates `CodeHint`'s algorithms. Both examples involve the GUI code using the Java Swing toolkit shown in Figure 2.1. Readers are also encouraged to watch a demo video of `CodeHint` at http://www.cs.berkeley.edu/~joel/codehint/.

2.1.1 User Perspective Example

A common task when writing GUI code is to detect clicks on a graphical tree of elements using code similar to that in Figure 2.1. A programmer might be unsure how to find the clicked element so she can use it. Unfortunately, as she does not know the API, she is not even sure what type this object has; it could be a node object, the data it represents, or the displayed string. As she does not even know the type of the expression she desires, it is difficult to find.

Using `CodeHint`, she can easily set a breakpoint where she wants to insert the code and then run the program, see a tree like the one shown in Figure 2.2, click on a node, and give a pdspec that expresses which node she clicked. `CodeHint` will then synthesize some candidate code snippets and show them to her. She can inspect these and choose the one she likes, or she can click on a different element in the tree and give a different pdspec to refine the set of candidates.

```
1 final JComponent tree = makeTree();
2 tree.addMouseListener(new MouseAdapter() {
3   public void mousePressed(MouseEvent e) {
4     int x = e.getX(), y = e.getY();
5     Object o = null;
6     // Get the menu bar or the clicked element.
7   }
8 });
```

Figure 2.1: Code that listens for clicks on a graphical tree of elements.

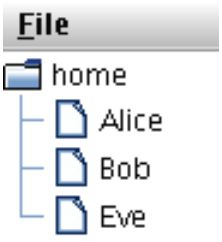


Figure 2.2: An example tree that might be created by the code in Figure 2.1.

To use `CodeHint` to find the code she desires, she can set a breakpoint after line 5 and then execute her code and click on an element to reach the breakpoint. Knowing that in Java most objects have a `toString` method that gives a string representation of their value, she realizes that if she clicks on an element labeled "Alice" (e.g., the one in Figure 2.2), the `toString` of her desired result should contain that string. This insight can lead her to enter the specification `o'.toString().contains("Alice")`. This specification, which we call a pdspec, encodes the fact that the value of the variable `o` should be updated by the desired statement so that its `toString` contains "Alice" (the `o'` denotes the value of the variable `o` after the code to be synthesized is evaluated). `CodeHint` will then generate eight expressions to assign to `o`:

```
((JTree)tree).getPathForLocation(x, y),
((JTree)tree).getSelectionPath(),
((JTree)tree).getLastSelectedPathComponent(), ...
```

To reduce the number of candidates, the user can continue the execution (or restart it) and click on a different element. Assume she does so and clicks on an element labeled "Bob". When the execution suspends at the breakpoint, `CodeHint` will show her all eight of the previous expressions with their results in the new context. She can then give a new pdspec to filter out some expressions. Giving `o'.toString().contains("Bob")` will remove one expression. Alternatively, by looking at the results of the eight expressions she might see that many return an object of type `TreePath` that seems to do what she wants, so she can

use the pdspec `o′ instanceof TreePath` to keep only those.

Assume that now the user clicks below all the elements (i.e., on nothing). She can then see all of the remaining candidate expressions with their new values and keep only those that evaluate to `null` with the pdspec `o′ == null`. `CodeHint` will then eliminate all but one candidate and find the correct code:

```
o = ((JTree)tree).getPathForLocation(x, y);
```

This example shows how our approach can be useful even when a programmer has very little knowledge about her desired result, perhaps not even its type, by allowing her to refer to values in the concrete program state. The programmer narrows down the set of candidate statements by incrementally providing pdspecs for different test scenarios. A key advantage of our methodology is that programmers can mix and match value demonstrations, type specifications, and arbitrary pdspecs as desired. In addition, the interactivity of `CodeHint` allowed the user to find the correct result from the initial candidates. This example was inspired by how one subject in our first user study solved this problem.

This example also shows how our approach can easily synthesize real code involving complicated libraries. While the final synthesized code appears simple, executing it involves making over 70 method calls that allocate new objects and use complex objects, generics, and binary-only libraries. Tools that rely on modeling the target language, e.g., to translate it to SAT, would thus have difficulty synthesizing this code.

Using skeletons. Perhaps now the user wants to get the data representing the object she clicked out of the `TreePath` object she just assigned to `o`. Let us assume that she changes `o`'s type in the code and discovers from the type's documentation that it contains a `getPathComponent` method that she thinks will help her. However, she is unsure what argument to pass to this method.

The user can encode this knowledge into a skeleton that `CodeHint` will use to guide its search. Specifically, she can enter the skeleton `o.getPathComponent(??)` where the `??` represents the missing portion of the code (she could also use `**` instead of `??` if she were unsure how many arguments were needed). Given this skeleton, `CodeHint` will search for expressions that can be used as arguments to `getPathComponent`, which expects an integer. This skeleton complements the user's pdspec: the skeleton determines the search space while the pdspec determines which results to show to the user.

As before, the user can set a breakpoint where she wants to insert code, run the program on a test, and click on an element to hit the breakpoint. If she clicks on "Eve", she can enter the pdspec `_rv′.toString().equals("Eve")` to show that she now wants some object that represents "Eve" (the `_rv′` represents the return value of the expression) along with the skeleton discussed above to guide the search.

`CodeHint` will now search for integers that, when passed to `getPathComponent`, meet the user's specification, which include the following:

```
o.getPathComponent(e.getClickCount())
o.getPathComponent(o.getPathCount() - 1)
```

By refining the set of candidates by giving different pdspecs in different states, the user can remove incorrect expressions such as the first one above.

This example shows how users can encode their partial knowledge of the correct code to allow `CodeHint` to focus its search on relevant code snippets.

2.1.2 Algorithm Example

We now walk through a similar example of using `CodeHint` where we focus on its algorithms rather than the user's interactions with it.

Imagine that a user writes the partial code shown in Figure 2.1 and then wants to write code at line 6 to find the menu bar for the window that contains the graphical tree and store it in the variable `o`. A simple Internet search will reveal that the menu is represented by a `JMenuBar` object but will likely provide little information on how such an object can be acquired.

To use `CodeHint` to find this object, the user can set a breakpoint after line 5 (e.g., near the comment on line 6) and run the program to that breakpoint, which is the current context. Since she knows that she wants to assign a value of type `JMenuBar` to the variable `o`, she can provide the pdspec `o′ instanceof JMenuBar` to `CodeHint` to encode the fact that `o` should change so that it contains an object of type `JMenuBar`.

Given this query, `CodeHint` will begin a search for expressions that it can assign to `o` to try to satisfy the pdspec. This iterative search will start with local variables and generate larger expressions with operations such as addition and method calls. `CodeHint` will evaluate these expressions in the current context, undoing side effects as they occur, to enable it to get precise results that satisfy the user's pdspec. A probabilistic model will guide the search toward more likely expressions and `CodeHint` will group equivalent expressions together to avoid duplicate work.

Once this search is complete, `CodeHint` will show the user approximately five results. She can then add a new testcase to the code and run that new input, which will allow `CodeHint` to remove two of the previous results that crash in the new context.

We now give more details on how this process works by walking through `CodeHint`'s algorithm and the user's interactions with it.

First iteration. `CodeHint` will first query the debugger for all the variables in scope, evaluating them so it knows their dynamic types. These, along with the special values `null` and `this`, will become `CodeHint`'s initial set of candidate expressions:

```
tree, e, x, y, o, this, null
```

Second iteration. In its next step, `CodeHint` will combine these simple expressions into more complicated ones according to the Java grammar. To do this, it will first query the debugger for the dynamic type of each candidate.

For each object, it will query the debugger for all accessible methods available on that type (and its supertypes) and then call each of those methods with all type-safe combinations of the previous set of candidates as arguments (downcasting when necessary).

As an example, `CodeHint` will find that `tree` is a value of type `JTree` (which is a subtype of `JComponent`, the static type of the variable). It will then ask the debugger for the methods of `JTree`, one of which is `getPathForLocation`. This method expects two integer arguments, so `CodeHint` will find all of the integer-valued expressions in its previous set of candidates, which in this case are `x` and `y`. `CodeHint` will then call this method with all possible combinations of these arguments, producing the following four calls:

```
((JTree)tree).getPathForLocation(x, x)
((JTree)tree).getPathForLocation(x, y)
((JTree)tree).getPathForLocation(y, x)
((JTree)tree).getPathForLocation(y, y)
```

`CodeHint` will repeat this process for other methods and for static methods of classes imported in the current file.

For each pair of primitives, such as integers, `CodeHint` will combine them with binary operations. Thus given `x` and `y`, it will generate the following expressions:

```
x + y, x - y, y - x, x * y, x / y, y / x,
x == y, x != y, x < y, x <= y, x > y, x >= y
```

It will similarly generate object comparisons, array accesses and length, field accesses, integer and boolean negation (e.g., `-x`), addition and subtraction with `1` (e.g., `x + 1` and `x - 1`), comparisons with `0` (e.g., `x > 0` and `x < 0`), and boolean conjunctions and disjunctions.

`CodeHint` will evaluate each expression as it is generated so that it knows its result. However, expressions with side effects must be handled correctly so they do not affect future evaluations. For example, `tree.add(makeTree())` will modify the tree, potentially causing future evaluations to return different results in this new context.

To avoid this problem, `CodeHint` uses novel techniques based on breakpoints and the Java security manager to undo in-memory side effects after they occur and block harmful native calls. Harmless native calls, such as reading from a file, proceed normally. In practice, we have found that allowing users to choose to relax these restrictions often gives correct results in less time. In this case, evaluating `tree.add(makeTree())` will modify a field of `tree`. This will trigger a breakpoint installed by `CodeHint`, which will log the change and undo it after the evaluation finishes.

Once this process is complete, `CodeHint` will have a new set of approximately 240 candidate expressions:

```
tree, e, x, y, o, this, null, x + y, x < y, tree.getTopLevelAncestor(),
Window.getWindows(), ((JTree)tree).getPathForRow(x), ...
```

The set of candidates can grow quite large, so `CodeHint` applies some optimizations that make it significantly smaller:

- To avoid repeating the same computations, `CodeHint` groups expressions into equivalence classes based on their results and only retains one representative of each class in its set of candidates. For example, `tree.getTopLevelAncestor()` is equivalent to `SwingUtilities.getRoot(tree)` in the current context, so only one (the former in this case) will be included in the set of candidates. As we will see shortly, `CodeHint` will use the equivalent expressions that are not in the list of candidates to help generate the results it shows to the user by substituting equivalent subexpressions.

- To avoid spending time searching expressions that are rarely used in practice, `CodeHint` uses a probabilistic model to avoid unlikely method calls and field accesses. This model contains information from over ten million real-world lines of code. As one example, the `JTree` class contains a method called `getNextMatch` that was not called once in all of the analyzed code, so `CodeHint` will not call it in this iteration.

`CodeHint` would have approximately 370 candidates without either of these optimizations and approximately 560 without both. Since these candidates are used to generate more expressions in the next iteration, these optimizations significantly improve performance, as we describe below.

None of these candidates meet the user's specification, as none have type `JMenuBar`, so `CodeHint` will automatically continue this process of creating larger expressions from its current candidates.

Third iteration. The third iteration proceeds exactly as the second: it combines the current candidates to produce larger expressions.

In this iteration, `CodeHint` will use its probabilistic model to avoid searching some additional expressions. Libraries such as Swing often contain many constants that are intended to be used only in certain contexts. For example, `KeyEvent.VK_ENTER` helps determine if the user pressed the Enter key. To avoid using such constants in unrelated contexts, `CodeHint`'s probabilistic model stores exactly how they are used in practice. This allows it to avoid generating expressions such as `tree.getComponent(KeyEvent.VK_ENTER)`, as it recognizes that `KeyEvent.VK_ENTER` was never used as an argument to `getComponent` in the analyzed code.

During this iteration, `CodeHint` will find that `tree.getTopLevelAncestor()`, whose static return type is `Container`, has type `JFrame` at runtime. `CodeHint` will then explore all the methods it can call on a `JFrame`, including `getJMenuBar`, which returns a `JMenuBar`. It will thus add `((JFrame)tree.getTopLevelAncestor()).getJMenuBar()` to its next set of candidates.

At this point, `CodeHint` will also explore some expressions that it avoided before due to its probabilistic model, such as calls to `JTree.getNextMatch`. This allows the algorithm to prioritize more likely expressions while remaining complete.

Since it calls each method with all possible type-safe combinations of the previous set of candidates, `CodeHint` might end up making an large number of calls to a single method. For

example, in this iteration it attempts to call the `getTreeCellRendererComponent` method of the `DefaultTreeCellRenderer` class. This method has seven arguments, including one of type `Object` (of which `CodeHint` has seen over 70 unique values), one of type `int` (of which `CodeHint` has seen over one hundred unique values), and four booleans (each of which can be `true` or `false`), which would result in well over 100,000 calls. To avoid spending so much time calling a single method, `CodeHint` only calls it a small number of times. Specifically, if a method can be called more than a certain number of times that grows exponentially with the current iteration, `CodeHint` calls it at most that many times with arguments that were candidates from previous iterations.

At the end of this iteration, `CodeHint` will have generated and evaluated over 700 expressions. Without either the equivalence class or the probabilistic model optimizations described above, `CodeHint` would have generated almost 7,000 expressions and therefore done an order of magnitude more work.

Once it has finished this process, `CodeHint` will find which of its candidates satisfy the user's pdspec. In this case, `((JFrame)tree.getTopLevelAncestor()).getJMenuBar()`, when assigned to `o`, is the only expression that satisfies the pdspec. Before showing it to the user, `CodeHint` will generate more satisfying expressions by replacing its subexpressions with equivalent expressions. For example, because `SwingUtilities.getRoot(tree)` is equivalent to `tree.getTopLevelAncestor()`, `CodeHint` knows without any additional evaluations that `((JFrame)SwingUtilities.getRoot(tree)).getJMenuBar()` will yield the same result and hence also satisfy the user's pdspec. `CodeHint` will thus generate this expression and three others that are all equivalent to the original expression.

Now that it has expressions to show to the user, `CodeHint` will use its probabilistic model to present the user with the more likely options closer to the top. A call's likelihood is the number of calls to it that the model contains and an expression's likelihood is the product of its subexpression's likelihoods. In this case, the `getTopLevelAncestor` method was called 19 times in the model while `getRoot` was called 15 times, so the former expression is shown first.

`CodeHint` will then show the user these expressions that, when assigned to `o`, meet the specification (as well as their values, side effects, and string representations):

```
((JFrame)SwingUtilities.getWindowAncestor(jtree)).getJMenuBar()
((JFrame)tree.getTopLevelAncestor()).getJMenuBar()
((JFrame)SwingUtilities.getRoot(tree)).getJMenuBar()
...
```

If `CodeHint` had not found any results that satisfied the user's pdspec at this point, it would have notified the user and asked if it should continue for another iteration.

Note that the results `CodeHint` found all rely on being able to discover the dynamic type of an expression and downcast to it, something that static tools will find difficult to do.

Refinement. The user can now examine these expressions, their results, and their documentation to try to pick the one she wants to use. If she is unsure which is correct, she can

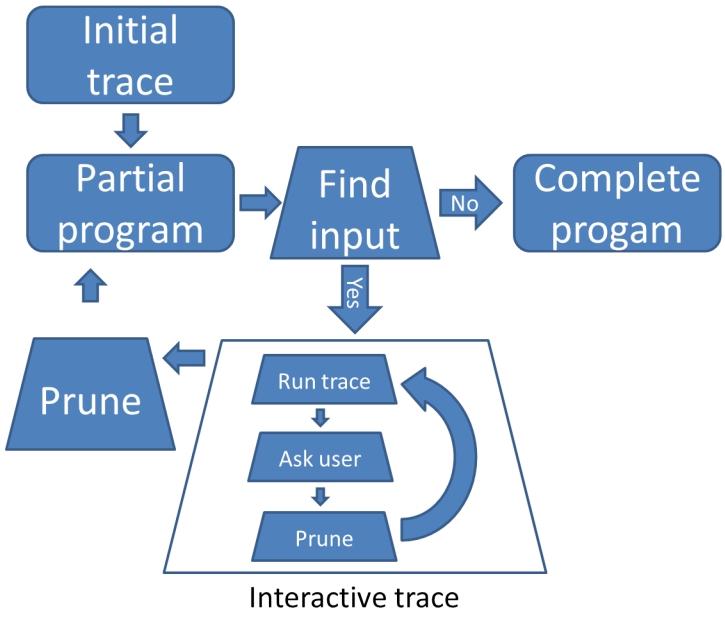


Figure 2.3: A high-level overview of our approach.

refine the set of results by running the program on a different input.

Let us assume that the user modifies the code so that the tree is contained within an applet that is itself contained within the top-level window, which might be useful for running the same code both by itself and within a web browser. Once she makes this change, she can run the new program until it reaches the point where she ran the previous search. `CodeHint` will then evaluate the previous results in the new context. In this case, two of them, including `((JFrame)tree.getTopLevelAncestor()).getJMenuBar()`, will crash. `CodeHint` will show this smaller set of results to the user, who may select one of them to use or continue the process with yet another input.

2.2 Programming by Demonstration

In our methodology, the user begins by giving an initial trace of the algorithm on a concrete input by stepping through and showing its execution. We transform this trace into a *partial program* that represents multiple programs that encompass the ambiguities in the trace. Using automated *pruning* techniques, we resolve as many ambiguities as we can. We then generate a new input that will resolve some of the remaining ambiguities and interactively execute it with the user, who resolves any ambiguities that are encountered and hence refines the partial program. During this *interactive trace* process we *infer control flow* that is missing from the user's traces. We continue this process as long as we can, at which point we present the final program to the user. We present a graphical overview of this approach in Figure 2.3.

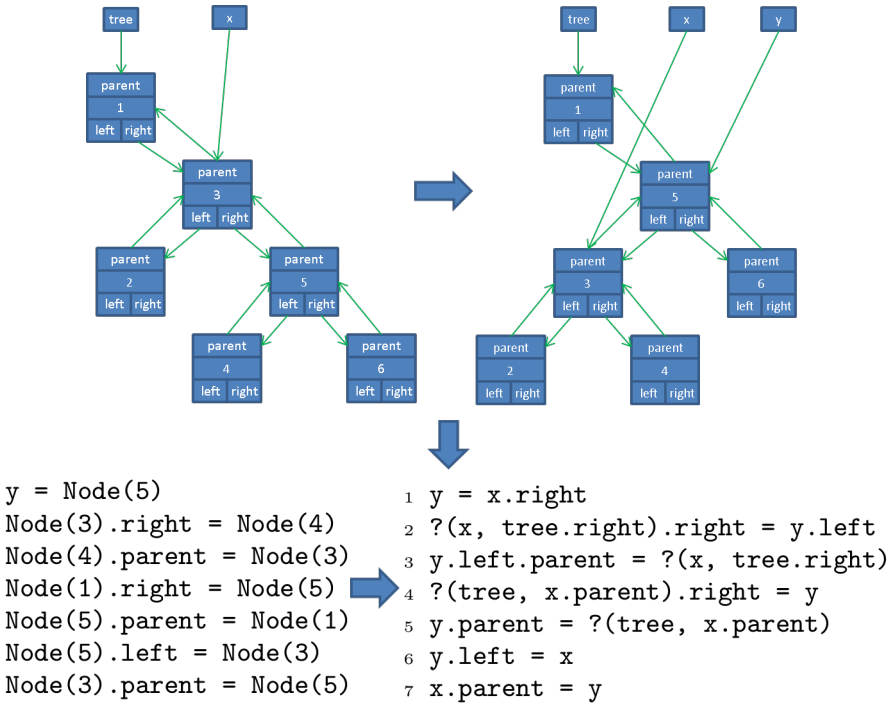


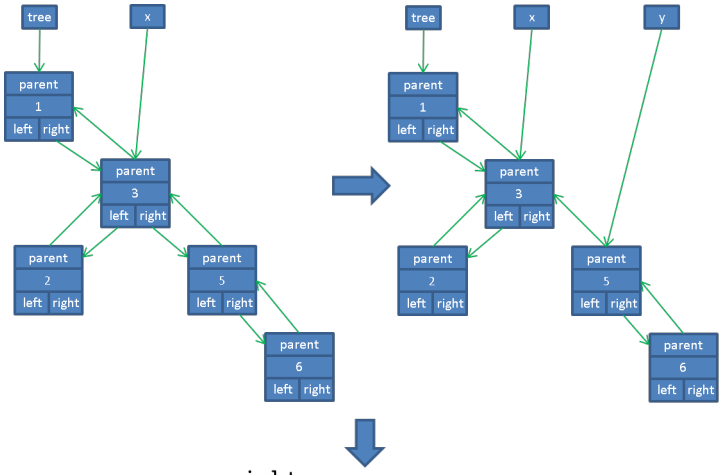
Figure 2.4: The first trace of a left rotation of a binary tree. The user is shown the left tree and manipulates it to reach the right tree.

We now demonstrate our approach by showing how it can be used to write code for a left rotation of a binary search tree. Various self-balancing binary search trees such as red-black trees [34] use rotations to improve their worst-case lookup time. A left rotation rotates a subtree counter-clockwise. The top part of Figure 2.4 shows a left rotation around the node pointed to by the variable `x`.

A user of our tool will first define the types his code requires and the signature of the function he is writing. He may then supply code that generates inputs to this function and checks that the output is correct. This correctness condition is not required but often proves to be helpful (see Section 5.2 for details). In our example, we assume the user writes a postcondition that specifies only that the output should be a valid tree (ignoring the red-black property) that contains the same elements as the input tree.

The user is shown a sample input to his function, such as the tree shown in the left of Figure 2.4. He then graphically manipulates the pointers (e.g., by dragging them to point to different objects) until he reaches the tree shown to its right, which is the correct output. Our tool creates a trace of this editing process, which is shown in the bottom-left of Figure 2.4.

It then synthesizes a partial program that could yield this trace. In certain cases, multiple different expressions might yield a value seen in the trace, so our tool creates a *hole* that represents the multiple possibilities for the desired expression. For example, due to aliasing, we are unsure whether the node with value 3 should be accessed via `x` or `tree.right`, and




```
1 y = x.right
2 ?(x, tree.right).right = y.left
3 if (y.left != null)
4   y.left.parent = ?(x, tree.right)
5 ?(tree, x.parent).right = y
6 y.parent = ?(tree, x.parent)
7 y.left = x
8 x.parent = y
```

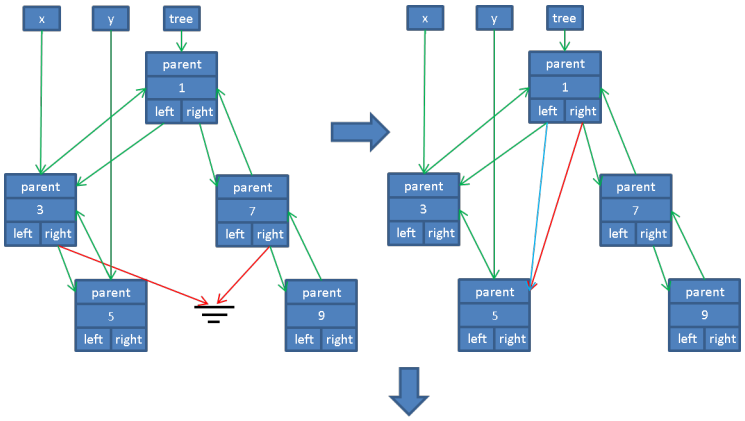
Figure 2.5: The second trace of a left rotation of a binary tree. The user is shown the left tree and manipulates it to reach the right tree.

similarly whether node 1 is `tree` or `x.parent`. The code in the bottom-right of Figure 2.4 is a simplified form of what this partial program looks like after seeing the above trace (the real version has many more holes).

Upon analyzing this partial program and running it on newly-generated inputs, our tool will discover that line 3 will dereference `null` on some inputs. It will thus present the user with such an input tree and ask him to give another trace and show how to avoid crashing. To make the user's task easier, the tool will suggest the next statement to execute by graphically showing it to the user, who can step forward as in a debugger.

Our tool might thus present the user with the input tree shown in the left of Figure 2.5. He can follow the tool's suggestions for the first two statements (`x` and `tree.right` are still aliased, so there is no ambiguity) to reach the tree shown to its right. At this point, our tool realizes that executing the next statement would crash, and so it suggests to the user that there might be a conditional at this point in the program. The user will recognize that `y.left.parent` should only be modified when `y.left` is non-`null`, and so will tell the tool this by demonstrating the condition (or its result) and the new branch. The partial program at this point is shown in the bottom of Figure 2.5.

Our tool will analyze this partial program and realize that all possible paths will fail the user's postcondition on some inputs, such as the one in the left of Figure 2.6. This is because




```
1  y = x.right
2  x.right = y.left
3  if (y.left != null)
4    y.left.parent = ?(x, tree.right)
5  if (x == x.parent.right)
6    ?(tree, x.parent).right = y
7  else
8    ?(tree, x.parent).left = y
9  y.parent = ?(tree, x.parent)
10 y.left = x
11 x.parent = y
```

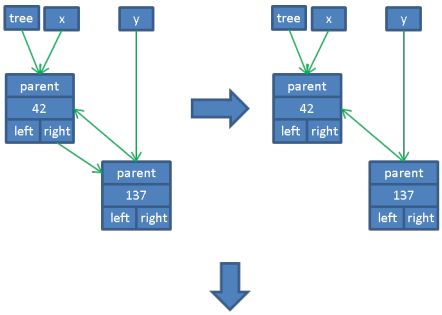
Figure 2.6: The third trace of a left rotation of a binary tree. The user is shown the left tree and manipulates it to reach the right tree. The red arrows represent the choices our tool shows the user and the blue arrows are the correct changes.

line 5 currently sets a `right` pointer to point to `y`, while the correct code might instead set a `left` pointer depending on whether `x` is a left or right child of its parent. For example, on this tree, the current code would change node 1's right pointer to point to node 5 (the red arrow in the right of in Figure 2.6), which would orphan nodes 7 and 9.

The tool will thus present an input such as the one on the left of Figure 2.6 to the user and ask him to walk through the algorithm. However, when executing line 2, it will recognize that there are two different possibilities, as `x` and `tree.right` represent two different objects. It will thus present both options to the user and ask which is correct (just as in the left of Figure 2.6); the user can simply click the correct option to resolve the ambiguity and continue.[1]

Upon reaching line 5, our tool will suggest changing node 1's right pointer to point to node 5 (with the red arrow in the right of Figure 2.6). The user will recognize that this is incorrect and tell the tool that there is a conditional at this point. Having now seen inputs

[1]In practice, our implementation will avoid asking the user this question by instead presenting him with a different input where it is not needed.




```
y = x.right
x.right = y.left
if (y.left != null)
  y.left.parent = x
if (x == tree)
  tree = y
else if (x == x.parent.right)
  x.parent.right = y
else
  x.parent.left = y
y.parent = x.parent
y.left = x
x.parent = y
```

Figure 2.7: The fourth trace of a left rotation of a binary tree. The user is shown the left tree and manipulates it to reach the right tree.

that go down both paths in the conditional, he realizes that the conditional must check whether `x` is a left or right child of its parent, and that in this case the left pointer must change (the blue arrow). The partial program after this new demonstration is shown in the bottom of Figure 2.6 and shows how our tool can infer conditionals.

Our tool will then realize that line 5 of this new partial program will dereference `null` when `x` has no parent and so will show the user a tree like the one shown in the left of Figure 2.7. The user will walk through the trace, accepting the suggested changes, until the execution reaches line 5, where a crash would occur. As before, the tool then suggests to the user that there is a conditional here, and the user will show the tool the correct action (to assign `y` to `tree`).

Our tool will be unable to find any failing inputs on the new partial program. It will then automatically generate new inputs and use them to prune incorrect possibilities and resolve the remaining ambiguities. For example, given inputs where `x` is the root of the tree, the `x.parent` possibilities will crash and so must be incorrect. Similarly, given a tree where `x` is not equal to `tree.right`, using the latter on line 4 will set the parent pointer incorrectly and hence fail the postcondition. This pruning allows our tool to generate the correct code shown in the bottom of Figure 2.7.

In this example, the user gave four traces before finding the correct code. Since the

code contained one conditional with three branches, the user must give at least three traces. We thus asked one trace more than the minimum required; with a different random seed, we might ask more or fewer questions. We asked the user to disambiguate among multiple possible actions once; as noted, we would likely not ask that in practice.

Chapter 3

Synthesis Algorithm

We now define pdspecs and our synthesis algorithm, which we use in both `CodeHint` and our PBD tool.

3.1 Approach

Without loss of generality, let us assume that we have an incomplete program in which a statement, say $s_T \in S$, is missing. Here S is the set of all syntactically valid statements. The developer wants to synthesize s_T using `CodeHint`. She creates a test input so that the execution of the program on the test input reaches the program location, say ℓ, that is just before the missing statement. Let σ be the program state when the program reaches the program location ℓ. We use Σ to denote the set of all feasible program states. The goal of `CodeHint` is to discover the missing statement s_T. Let $\text{exec}(\sigma, s) \in \Sigma \times S \to \Sigma$ be the program state obtained by executing the statement s in state σ. Let us use σ' to denote $\text{exec}(\sigma, s_T)$, i.e., the program state reached after the user executes the missing statement s_T in the state σ. As a running example, let us assume that σ is $\{x \mapsto 42\}$ and s_T is `x = 2 * x`. Then σ' is $\{x \mapsto 84\}$.

The user does not know the missing statement s_T, but she might have a good idea about what the program state should look like after the execution of s_T (i.e., σ'). With `CodeHint`, the user can indirectly provide a hint about the statement s_T by giving information about the state σ' using pdspecs. A pdspec can give absolute information about σ' by describing the updated program variables and their corresponding values or it can be a predicate relating the states σ and σ'. For example, a pdspec can specify `x' == 84`, where `x'` represents the value of `x` in state σ', or it could specify `x' > x`, a predicate that relates the states σ and σ'. In general, a pdspec can consist of any expression in the host language.

Note that a pdspec that is specified in the state σ may not be a correct pdspec if the program is in a different state at the location ℓ. (A different state at location ℓ can be reached by executing the program on a different test input.) For example, `x' > x` is not a correct pdspec if the program state at location ℓ is $\{x \mapsto -20\}$, but it is a correct pdspec

if the state is $\{x \mapsto 10\}$. Similarly, `x' == 84` is not a correct pdspec for any program state with `x != 42`. Note that `x' == 2 * x` is a correct pdspec for all program states that reach the location ℓ.

Formally, a pdspec is a logical predicate $\phi \in \Sigma \times \Sigma \to \{\text{true}, \text{false}\}$ where $\phi(\sigma, \sigma')$ checks whether σ' is a desired output state given the input state σ. As a notational convenience, for a pdspec $\phi(\sigma, \sigma')$, we refer to variables in the input state σ using their names and variables in the output state σ' using their primed names. All variables not given in a pdspec must be equal in the two states and all expressions in a pdspec must be free of side effects.

The user of `CodeHint` gives a sequence of pairs of program states at ℓ (reached by executing the program on different test inputs) and their corresponding correct pdspecs: $(\sigma_0, \phi_0), (\sigma_1, \phi_1), \cdots$. Based on the sequence of pairs of states and pdspecs, `CodeHint` returns a set of candidate program statements that could replace the missing statement. For example, if the user provides the sequence of pairs $(\sigma_0, \phi_0), \cdots, (\sigma_n, \phi_n)$, the set of suitable statements for the location ℓ is reduced to the set

$$C_n = \left\{ s \in S : \bigwedge_{0 \le i \le n} \phi_i(\sigma_i, \text{exec}(\sigma_i, s)) \right\}.$$

The predicate $\phi_i(\sigma_i, \text{exec}(\sigma_i, s))$ is true if the state σ_i and the state obtained after executing statement s in the state σ_i satisfy the pdspec ϕ_i. Statement s is in the candidate set if this predicate is true for all i. If a statement s is in the candidate set, it implies that at least for the program states $\sigma_1, \cdots, \sigma_n$ observed at the location ℓ, the execution of the statement s will result in the user's expected program state. If we could compute the candidate set for all possible pairs of programs states reachable at ℓ and their corresponding correct pdspecs, the set is guaranteed to contain the target statement s_T. However, in practice it not possible to enumerate all such possible pairs. Instead, the user demonstrates such pairs one-by-one and `CodeHint` computes a candidate set from the pairs provided by the user. Note that $C_{n+1} \subseteq C_n$, so the size of the candidate set shrinks as `CodeHint` receives more pairs. At any point, if the user notices a statement she could use as a substitue for the missing statement, she stops the process and uses the statement.

As the set of legal statements S could be infinite, `CodeHint` restricts it to a finite set F. For example this set could only include statements whose abstract syntax trees have height at most k and only use the variables available in the current scope. We call k the depth of the search space.

In practice, users often have some idea of the structure of the statement they desire. In our running example, the user might know that she wants to multiply `x` by something but might not know exactly what should be multiplied. By providing a *skeleton* (discussed more in Section 3.2), the user can further restrict the search space F into F_H. Here, this skeleton might be `x = ?? * x`, where `??` could be replaced by any valid expression.

In our running example, given the initial state $\{x \mapsto 42\}$, the pdspec `x' > 42`, and the skeleton `x = ?? * x`, we present the user with the following set of candidate statements:

$C_0 = \{$ `x = 2 * x, x = x * x,` $\cdots$ $\}$

Without the skeleton, we would additionally have included statements like `x = 84`, `x = x + 1`, and `x = x + x`. In either case, the number of candidate statements might be large but will be finite.

Given another initial state $\{x \mapsto 6\}$, the pdspec `x' == 12`, and the same skeleton, we present the user with the subset $C_1 = \{$ `x = 2 * x` $\}$.

3.1.1 Properties

While s_T is the missing statement that the user is trying to find, there might be other statements that also model the user's desired behavior, such as `x = x + x`. We define S^* as the set of all such statements. These statements need only be equivalent to each other in states in which they can actually be executed. We thus define the notion of *valid* states, which are those in which the user implicitly expects statements in S^* can be executed. We refer to the set of all valid states as Σ_{S^*}.

To reason about the equivalence of statements, we define an equivalence relation $\sim_\sigma$ on statements in a given program state where $s_1 \sim_\sigma s_2$ iff $\text{exec}(\sigma, s_1) = \text{exec}(\sigma, s_2)$. With this definition, we can say that any two statements in S^* are equivalent under all valid initial states:

$$\forall s, s' \in S^*.\forall \sigma \in \Sigma_{S^*}.s \sim_\sigma s'$$

Continuing our running example, `x = x + x` is equivalent to s_T. In addition, if s_T can never be executed when `x` is negative, then `x = (x < 0 ? 5 : x + x)` is also equivalent to s_T.

A user might mistakenly provide a pdspec that does not hold for the statement s_T, such as `x' == x + 2`. We say that a pdspec ϕ is *valid* if it is a correct description of the user's desired behavior (i.e., that of the statements in S^*) in a given valid state. A pair (σ, ϕ) is valid, denoted $V(\sigma, \phi)$, iff both σ and ϕ are themselves valid, i.e.,

$$V(\sigma, \phi) = \sigma \in \Sigma_{S^*} \wedge \forall s \in S^*.\phi(\sigma, \text{exec}(\sigma, s)).$$

If a user gives an invalid pdspec that does not hold for any of the statements in S^*, none of those statements will be in the corresponding C_n. However, if all of his pdspecs are valid, C_n will include all statements in S^* that are also in F_H.

Lemma 1 (Qualified completeness)**.**

$$\forall 0 \leq i \leq n.V(\sigma_i, \phi_i) \Rightarrow S^* \cap F_H \subseteq C_n$$

Proof. Let $s \in S^* \cap F_H$. For all $0 \leq i \leq n$, since $V(\sigma_i, \phi_i)$ holds by assumption, $\phi_i(\sigma_i, \text{exec}(\sigma_i, s))$, and so $s \in C_n$. □

In general, C_n might contain statements that do not have the behavior desired by the user on all states. The user can remove such statements from subsequent C_n by providing well-chosen input states or pdspecs.

Lemma 2 (Soundness)**.**

$$\forall s \notin S^*.\exists 0 \leq i \leq n.\neg\phi_i(\sigma_i, \text{exec}(\sigma_i, s)) \Rightarrow C_n \subseteq S^*$$

Proof. Let $s \in C_n$. If $s \notin S^*$, then by assumption $\exists 0 \leq i \leq n.\neg\phi_i(\sigma_i, \text{exec}(\sigma_i, s))$. Then by definition $s \notin C_n$, which is a contradiction, so $s \in S^*$. □

We can combine our notions of soundness and completeness to specify when we find exactly the statements in $S^* \cap F_H$.

Theorem 3 (Qualified correctness)**.**

$$\forall s \notin S^*.\exists 0 \leq i \leq n.\neg\phi_i(\sigma_i, \mathit{exec}(\sigma_i, s)) \wedge \forall 0 \leq i \leq n.V(\sigma_i, \phi_i) \Rightarrow C_n = S^* \cap F_H$$

Proof. Follows from Lemmas 1, 2, and the fact that $C_n \subseteq F_H$. □

We note that our formalism could be extended to search for sequences of statements by considering a block of statements as a single statement. It can also extend to search expressions by generating an assignment to a fresh temporary variable.

3.1.2 Classification of pdspecs

As we saw in the previous section, a valid pdspec only needs to hold for the desired statement in the current state. It does not necessarily need to reject all undesired statements or hold in all states. We now define and discuss these properties in more detail.

As logical formulae, it makes sense to talk about the *strength* of a pdspec using logical implication. A pdspec ϕ is clearly stronger than ϕ' if $\phi \Rightarrow \phi'$. However, our pdspecs are given in the context of a program state and are used to refine the C_n generated by previous states and pdspecs. It thus seems natural to compare the strength of pdspecs by the number of statements in the current set of candidates that satisfy them in the given state.

Formally, we say that a pdspec ϕ is stronger than ϕ' in a state σ if, given a set of candidates C_n,

$$|\{s \in C_n : \phi(\sigma, \text{exec}(\sigma, s))\}| \leq |\{s \in C_n : \phi'(\sigma, \text{exec}(\sigma, s))\}|.$$

We note that if $\phi \Rightarrow \phi'$ then ϕ is stronger than ϕ' in all states.

If the user gives a stronger pdspec, the corresponding set of candidates C_n will be smaller, which means that the user will need fewer refinement iterations before finding the desired statement. However, even weak pdspecs can be helpful as the number of refinements n increases.

Some pdspecs, such as `x' == x + x` above, are valid for all valid states, while others, such as `x' > 42`, are only valid for certain input states. We thus define the *context-dependence* of a pdspec, which describes how much it depends on the corresponding state. Formally, we

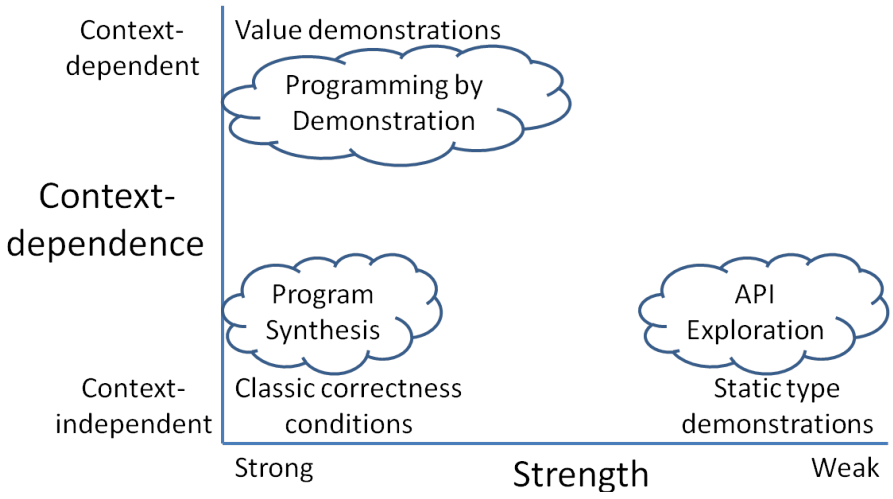


Figure 3.1: The spectrum of pdspecs that we allow, with some special cases.

say that a valid pdspec ϕ is more context-dependent than a valid pdspec ϕ' if it holds in fewer valid states:

$$|\{\sigma \in \Sigma_{S^*} : V(\sigma, \phi)\}| \leq |\{\sigma \in \Sigma_{S^*} : V(\sigma, \phi')\}|.$$

The concepts of strength and context-dependence define a spectrum of pdspecs, which is depicted in Figure 3.1. Other pdspecs, such as `o'.toString().contains("Alice")` from Section 2.1.1, help fill out the center of this diagram.

3.1.3 Relation to Previous Work

The classifications of pdspecs we developed in Section 3.1.2, which is graphically depicted in Figure 3.1, can help us explain how our approach relates to previous work. By allowing a wide variety of specifications that previous work has shown to be useful, we believe our pdspecs will be helpful to users.

Work on Programming by Demonstration [39, 24, 55, 50] synthesizes statements from programmer-provided demonstrations of their results. These demonstrations correspond to value demonstrations such as `x' == 42` that are stronger than all other valid pdspecs and (usually) very context-dependent, and so Programming by Demonstration fits in the upper-left part of Figure 3.1.

Programmers using tools from program synthesis [81, 48, 47] can give functional specifications of their desired program and get back code that is synthesized using techniques such as SMT solvers. As these specifications often need to be quite strong for the synthesis to be successful, these approaches often fall in the lower-left part of Figure 3.1.[1]

[1]These techniques usually give their specifications statically; we note that we could allow programmers to give completely context-independent pdspecs without an explicit execution context.

Research on exploring new APIs [59, 44, 74, 88] generally finds code given queries that specify the desired output type. Our pdspecs can easily encode these specifications, e.g., `x′ instanceof JMenuBar`. These queries are often fairly weak and context-independent, placing this body of work in the bottom-right part of Figure 3.1.

3.2 Algorithm

We have developed two versions of our algorithm: a deterministic version and a stochastic version. We begin by explaining the pieces that both share before describing them in detail in Sections 3.2.1 and 3.2.2. While the general approach we describe here can synthesize arbitrary statements, our current implementations of both versions only use them to synthesize expressions (and simple assignments using them). We describe how our current implementations can synthesize more complex statements in Sections 4.1 and 5.3.

Our algorithm begins by collecting all the variables in scope and then iteratively uses the current set of statements to generate larger ones. When it reaches a fixed maximum size, it removes those that do not satisfy the user's pdspec.

A key insight is that if two statements have the same effects, we can treat them as equivalent when generating further statements. We take advantage of this fact by grouping statements into equivalence classes based on their effects and values. As an example, there might be hundreds of pure boolean-valued expressions at a given depth, but they all evaluate to either `true` or `false`.

We define an equivalence relation on statements such that two statements are equivalent if they have the same side effects (and yield the same value if they are expressions) in the current state. We write $s_1 \sim_\sigma s_2$ to denote that the statements s_1 and s_2 are equivalent in the state σ.

We take the current set of side effects into account when generating new statements from representatives of these equivalence classes. For example, when `x` is `42` we will notice that `x` and `42` are equivalent. Later, when considering expressions to add to `x++`, we will use the equivalence class for the state where `x` is `43` and consider `x` and `42` separately.

Later in the search we can use the equivalence classes to generate extra statements without executing them by replacing equivalent substatements. For example, if `x` and `z.f(0)` are equivalent, then once we generate `p.bar(x)` we can automatically generate `p.bar(z.f(0))` as well.

We apply a number of optimizations to improve the efficiency of our algorithm. We use a variety of simple structural techniques to avoid enumerating obviously equivalent expressions (such as `x+y` and `y+x` for integers `x` and `y`). We cache the result and effects of each statement and use them instead of the statement itself in future evaluations to avoid duplicating work.

By actually executing the generated statements, our algorithm can synthesize real-world Java code, including file I/O, binary libraries, reflection, foreign function calls, and calls to

the user's own methods.[2] Thus while the individual statements we generate appear somewhat simple, they can in fact be quite complicated. In addition, we are able to synthesize code for Android without any extra modeling.

Probabilistic models. To guide our algorithm toward exploring more likely statements, we have added a probabilistic model based on an offline analysis of over ten million lines of code. This allows us to compute how often certain types, methods, and fields are used. Having analyzed how often each method and field is accessed, we define the probability of accessing member `m` (calling a method or accessing a field) of type `T` as

$$\begin{aligned} P(m) = P(m|T)P(T) &= \frac{\#\text{ accesses of } m \text{ on } T}{\#\text{ of accesses on } T} \times \frac{\#\text{ of accesses on } T}{\#\text{ of accesses}} \\ &= \frac{\#\text{ accesses of } m \text{ on } T}{\#\text{ of accesses}}. \end{aligned}$$

To smooth this model, if the type `T` had not been seen before, we gave the access the average probability of all accesses in the model, and if `T` had been seen but `m` had not, we gave it the probability $\frac{1}{\#\text{ of accesses}}$. This model is somewhat simplistic, as it assumes that all method calls are independent, but we have found it very helpful in practice. The probability of a statement is the product of the probabilities of its individual accesses.

In addition, as Java classes often contain many constants, we store, for each constant field, all the places it is used as an argument to a method. Then the probability of using constant c as the i^{th} argument to method m is

$$\begin{aligned} P(c, m, i) = P(c, m, i|c)P(c) &= \frac{\#\text{ uses of } c \text{ on method } m \text{ at index } i}{\#\text{ of uses of } c} \times \frac{\#\text{ of uses of } c}{\#\text{ of uses}} \\ &= \frac{\#\text{ uses of } c \text{ on method } m \text{ at index } i}{\#\text{ of uses}}. \end{aligned}$$

We use this probabilistic model to guide the search as well as to sort the candidates presented to the user.

Skeletons. As mentioned earlier, users may provide a skeleton to shape the search space explored and guide it toward candidates they know to be likely. Skeletons consist of normal Java code with *holes* for unknown code that should be synthesized. The language contains two types of holes: simple holes (denoted `??`) and list holes (denoted `**`). Simple holes can take the value of any expression or name in the language or be annotated with a set of candidates. List holes are used for calling functions with an unknown number of arguments, each of which is a separate expression hole. Users may additionally check boxes indicating whether or not constructor calls and infix/prefix operators should be searched.

[2]Some of these features require the user to allow searching native calls, in which case we cannot undo side effects.

Algorithm 1 Deterministic synthesis algorithm.

Initialize *classes* with `null`, `this`, and local variables.
for $i \leftarrow 2, maxDepth$ **do**
  $newExprs \leftarrow$ genOneLevel($getRepresentatives(classes)$)
  Evaluate $newExprs$
  Add $newExprs$ to $classes$
end for
$result \leftarrow checkPdspec(getRepresentatives(classes))$
$result \leftarrow expandEquivalences(result)$

To give some examples, `??.??` represents accessing some field of an unknown expression, `foo.??{bar,baz}(??)` represents calling either the `bar` or the `baz` method of the `foo` object with a single unknown argument, and `??(**)` represents calling an unknown method with any number of unknown arguments.

While not as general as other similar languages (e.g., [65]), our skeleton language has the advantage of being simple and easy to use. Users need only remember two pieces of syntax: they can put `??` wherever they are unsure of an expression or a name and `**` when calling a method with an unknown number of arguments.

Since the given pdspec applies to the whole statement, we must explore the cross product of all the specified holes. Given a skeleton, we fill each hole with type-correct values using the algorithm described above (unless the hole has been annotated with candidate values, in which case we simply try those values in lieu of a search). If a skeleton has many holes, we reduce the depth of our search for each hole.

3.2.1 Deterministic synthesis algorithm

Our deterministic synthesis algorithm, shown in Algorithms 1 and 2, searches the space of possible expressions in a manner similar to breadth-first search. In particular, it generates expressions in increasing order of *depth*, which is the height of the parse tree. As examples, `x` has depth 1, `foo.bar(x,y)` has depth 2, and `foo.bar.baz(x+y)` has depth 3.

As shown in Algorithm 1, our algorithm iteratively builds all expressions up to a fixed depth, evaluating them with a timeout and creating equivalence classes as it goes. For each equivalence class, we use only one representative to generate further expressions. For example, if `x` and `z.f(0)` are equivalent, we generate `x+1` and `foo(x)` but not `z.f(0)+1` or `foo(z.f(0))`.

Algorithm 2 shows how a single iteration generates larger expressions. For each expression seen so far, it constructs larger expressions using it based on its type. For numbers, it generates infix operations with all numbers, and it indexes into arrays with all numbers. For objects, it generates all field accesses and calls all methods with all correctly-typed combination of the expressions seen so far.

Algorithm 2 A portion of one level of the deterministic synthesis algorithm, which works similarly to a single iteration of a breadth-first search.

```
procedure genOneLevel(exprs)
    newExprs ← ∅
    for all e ∈ exprs do
        if e is a number then
            for all numbers o ∈ exprs do
                newExprs ← newExprs ∪ {e + o, e − o, ···}
            end for
        end if
        if e is an array then
            newExprs ← newExprs ∪ {e[i]|i ∈ exprs, i is a number}
        end if
        if e is an object then
            newExprs ← newExprs ∪ {e.id|id ∈ fields(e)}
            for all methods m of e do
                newExprs ← newExprs ∪ {e.m(e1, e2, ···)|ei ∈ exprs, ei : argType(m, i)}
            end for
        end if
    end for
    return newExprs
end procedure
```

When the desired depth has been achieved, we test the equivalence classes for the initial state against the pdspec, filter out those that do not satisfy it, and recreate the full set of candidate expressions from the equivalence classes. Once we discover that an expression satisfies the pdspec, we know that all expressions generated from it by replacing its subexpressions with expressions that are equivalent in the current state will also satisfy the pdspec.

We use the probabilistic model to avoid calling rare methods and using constants in places they were rarely used by artificially increasing the depth of expressions that include them.

The size of the space of expressions that our algorithm must search at a given depth is exponential due to method calls. If at one point during a search we have seen 50 different integer values, there will be 50^3 different ways to call a method with three integer arguments. Such growth can easily overwhelm our algorithm, so we heuristically prune calls to methods that would otherwise be called with a large number of different arguments, especially ones our probabilistic model defines as uncommon. That is, if a method can be called more than a certain number of times that is exponential in the current depth, we artificially increase the depth of calls to it. As we will see in Section 4.3.1, such calls occur infrequently in practice. We believe this would be a promising area in which to integrate symbolic techniques.

Algorithm 3 Stochastic synthesis algorithm.

```
Initialize classes with null, this, and local variables.
for i ← 1, maxIters do
    expr ← choose an expression from classes
    newExpr ← extendExpression(expr)
    Evaluate newExpr
    newExprs ← expandEquivalences(newExpr)
    Add newExprs to classes
end for
result ← checkPdspec(getRepresentatives(classes))
```

Properties. Unlike symbolic techniques, our algorithm cannot "look inside" method calls and other operations and must treat them as black boxes. It is optimal given this restriction, as thanks to the equivalence classes, each expression it retains yields a unique value and could potentially be the only expression to satisfy the pdspec.

The number of expressions at a given depth is exponential because of method calls (although we address this problem below) and the total number of expressions is exponential in the depth of expressions. Despite this, we show in Section 4.3.1 that our algorithm performs well in practice and has synthesized up to ten lines of code at a time.

An additional benefit of our synthesis algorithm is that each iteration of the search should be easier to parallelize than SAT-based approaches, which require careful work to obtain parallel speedups.

3.2.2 Stochastic synthesis algorithm

Unlike the breadth-first approach used by our deterministic algorithm, our stochastic search algorithm, shown in Algorithms 3 and 4, uses a best-first search. Specifically, it uses the probabilistic model described above with some additional components to choose a single expression to extend each iteration and to choose how to extend it.

Note the differences between the stochastic search in Algorithm 3 and the deterministic search in Algorithm 1. The two run a different number of iterations, as each iteration in the stochastic search evaluates a single expression while iterations in the deterministic search evaluate many expressions. The deterministic search extends all expressions in a single iteration while the stochastic search extends only one. The deterministic search only expands equivalent subexpressions at the end of the search (when it knows exactly which ones it needs to expand) while the stochastic search expands after each new expression is evaluated.

The algorithm for expanding equivalent subexpressions is slightly different in the stochastic search. Given the newly-evaluated expression, it begins by generating expressions equivalent to it by replacing its subexpressions with equivalent ones. It then expands expressions that contain expressions equivalent to these expressions.

Algorithm 4 A portion of one level of the stochastic synthesis algorithm.

```
procedure EXTENDEXPRESSION(e, classes)
    if e is a number then
        o ← choose a number from classes
        ⊕ ← choose an operator from {+, −, ···}
        return e ⊕ o
    else if e is an array then
        i ← choose a number from classes
        return e[i]
    else if e is an object then
        id ← choose a method or field of e
        if id is a field then
            return e.id
        else
            (e_1, ···) ← (choose an expression from classes of type argType(m, 1), ···)
            return e.id(e_1, e_2, ···)
        end if
    end if
end procedure
```

The `choose` operator in Algorithms 3 and 4 chooses an element from a set based on their probabilities. For example, if the set is $\{e_1, e_2\}$ with $P(e_1) = 2P(e_2)$, then e_1 is twice as likely as e_2 to be chosen. We implemented this with a modified version of roulette wheel selection [32, 1] that computes the probabilities of each element, chooses a random number between 0 and their sum, and then uses binary search on the prefix sum to find the index of the chosen element. Formally, given elements $e_1, e_2, \cdots, e_n$, we compute the prefix sums of their probabilities $P(e_1), P(e_1) + P(e_2), \cdots, \sum_{i=1}^{n} P(e_i)$, choose a number in the range $[0, \sum_{i=1}^{n} P(e_i))$, and then use binary search to find the right element.

We chose this selection algorithm due to its incremental nature. During the search we constantly update the set of expressions we can extend (each iteration generates a new expression and adds it to this set). With this algorithm, adding a new element takes constant time, as we simply add it and its probability to the end of the list.

We make the following changes and additions to the probabilistic models described above.

New weighting. Previously, the probability of an expression was computed as the product of the probabilities of its subexpressions. Given that the probability of any individual expression is very small, this causes almost all larger expressions to have much smaller probability than almost all smaller expressions. For the deterministic algorithm this suffices, as it simply means that shorter expressions are placed higher on the list shown to the user. But it would mean that the stochastic search would mimic a breadth-first search, as the probabilities would cause it to prefer shorter expressions to larger ones.

To avoid this undesired behavior, the stochastic search computes the probability of an expression based on the average probabilities of its subexpressions. Doing only this would bias the search too much toward larger expressions made up of common subexpressions, so it also divides this probability by a function of the size of the expression. Given an expression made up of n subexpressions, our implementation currently computes its probability as the average probability of its subexpressions divided by 2^n. This allows the search to prioritize more common subexpressions while still preferring smaller expressions.

Form model. The form model computes the probabilities of different AST nodes used in expressions. For example, it computes the probability of a method call compared to a field access. It also computes the probabilities of different operators, e.g., `*` and `!`, and the probability of a static method call compared to an instance one.

The algorithm uses this model to ensure that the expressions it searches roughly match those used in practice. For example, when it is going to extend a number, its choice of whether to use a unary or a binary operator and which operator to use are both based on this model's probabilities. It also chooses whether to extend an object, array, or number based on this model's probabilities of operations on objects (field and method accesses), numbers (infix and prefix operators), and arrays (array accesses and length).

Type model. The type model ensures that the search encounters a variety of different types instead of simply focusing on the most popular ones. For example, many methods return an `int` or a `boolean`, but the latter will likely not help once we have seen both `true` and `false`, and searching an extra integer-valued expression is more useful when the search has encountered five different `int`s than when it has encountered 100.

Specifically, given a query for an expression with type `T`, the type model weighs a method or field based on its type `S` in the following order.

1. `S` is a subtype of `T`
2. `S` is a supertype of `T`
3. Given two types, prefer the one for which the search has encountered fewer values.

Thus the model prefers expressions that return `T` or a subtype of `T`, then those that return a supertype of `T`, then those for which it has encountered fewer values. As a special case, `boolean`s are given a very low weight after both `true` and `false` have been encountered.[3] As an example, when searching for type `JMenuBar`, the search prefers expressions of that type to those of type `Object`, which are preferred over those of type `IUnicorn` (or some other rare type), which are preferred over `int`s, which are preferred over `boolean`s once both `true` and `false` have been seen.

[3]Note that we still might want to search `boolean`s in this case because they could be the user's desired expression or have useful side effects.

Chapter 4

CodeHint

We begin in Section 4.1 by describing our implementation of `CodeHint`, which is based on the techniques we have just described. We then discuss the various probabilistic models we implemented while exploring different implementation strategies in Section 4.2. Finally, we evaluate the performance and productivity benefits of `CodeHint` in Section 4.3.

4.1 Implementation

We have developed an implementation of our approach, which we call `CodeHint`, as a plugin for the Eclipse IDE for Java. This allows users to develop normally, using our approach only when they wish to do so. `CodeHint`, its source, and video demos are available at https://github.com/jgalenson/codehint. Figure 4.1 provides an example of `CodeHint`'s interface that shows the user's pdspec and the results `CodeHint` finds.

To use `CodeHint`, a programmer must start a debug session and navigate to the program location and state in which she wishes to insert code. She then enters a pdspec and possibly a skeleton through a dialog. There are also shortcuts to demonstrate a value (by giving an expression and evaluating it) and the desired dynamic type.

`CodeHint` then uses the deterministic synthesis algorithm from Section 3.2.1 to synthesize candidate statements from the grammar shown in Figure 4.2, which includes variables, array accesses, casts, field accesses, method calls, constructor calls, and unary and binary operators, including calls to static methods and fields of imported classes.[1] It shows them, their results and `toString`s, and their side effects to the user, who can select which to keep. The user may also view the Javadocs of methods and fields used in the candidates and sort or filter them. If she does not find any expressions she wants, she can continue the search with an increased depth.

We wrap the expressions selected by the user in a call to a special `choose` method, which yields a source code representation of the set of statements C_0 from Section 3.1. When

[1]There seems to be no technical barrier to extending this language to cover all Java expressions (except for closures and anonymous class declarations), but we have seen no need to do so yet.

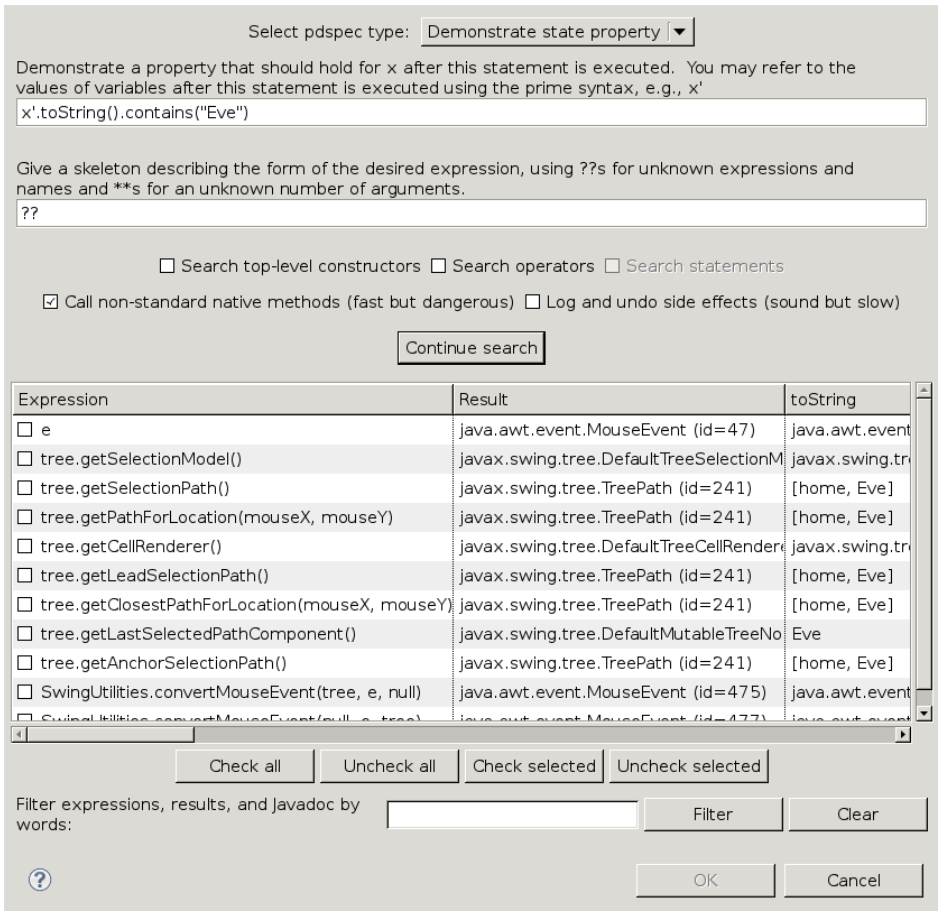


Figure 4.1: The `CodeHint` GUI on the example from Section 2.1.1.

there is only a single element, we replace the `choose` call with a call to a `chosen` method (which has identical semantics) as an indicator to the user that the process is complete. Figure 4.3 shows our Java implementation of these two methods (with a few minor details omitted). This simple runtime library allows the program to remain executable at all times so it can be used outside of our environment. When used inside our environment, the `choose` method triggers the UI for refinement; when used without the plugin it asserts that all of its arguments are equal.

Since the candidate expressions are inserted directly into the code, the user can directly modify C_n at any time. For example, she can simply select the desired expression and delete everything else, or she can remove expressions that she knows are incorrect so they are not considered again. Users who are only interested in the output with a particular set of inputs – of a one-time script, perhaps – may never need to find the exact statement.

After selecting the desired candidate expressions, the user can end the debug session and continue with a new input (perhaps from a different test case). She may also continue

(demonstration)	d		
(type)	t		
(local/arg)	v		
(identifier)	id		
(constant)	c	::=	`0` | `1` | `null`
(infix op)	$\oplus$	::=	`+` | `-` | `*` | `/` | `&&` | `||` | `==` | `!=` | `<` | `<=` | `>` | `>=`
(expression)	e	::=	d | c | `this` | v | $e \oplus e$ | `-`e | `!`e | e`[`e`]` | $e.id$ | $t.id$ | $e.id(e,\cdots)$ | $t.id(e,\cdots)$ | `new` $t(e,\cdots)$ | `(`t`)`e | `(`e`)`
(assignment)	a	::=	e`.id =` e | v `=` e
(statement)	s	::=	e`;` | a`;` | `{` s s $\cdots$ `}`

Figure 4.2: The grammar of expressions and statements `CodeHint` can generate.

```
public class CodeHint {
  public static <T> T choose(T first, T... rest) {
    for (T choice : rest) // We omit the special case when rest is null.
      assert first == null ? choice == null : first.equals(choice);
    return first;
  }
  public static <T> T chosen(T choice) {
    return choice;
  }
}
```

Figure 4.3: Our implementation of the `choose` and `chosen` methods (with some minor details omitted).

execution by using the value of one of the candidates. In practice, this works well for rapidly refining statements that are contained in loops or are otherwise executed multiple times.

Repeat encounters with an inserted `choose` statement are handled by inserting a breakpoint at the line and registering a listener that activates when that breakpoint is hit. This handler overrides the default implementation of `choose` given in Figure 4.3 and shows the user the previously-chosen candidates and their values in the new context. She may enter a new pdspec to refine the current set of candidates or she may abandon the current results and start a new search, which can be useful if she previously gave an invalid pdspec or if our search did not find any correct statements.

Java algorithm details. Following standard Java idioms and practices, two values $\mathtt{v}_1$ and $\mathtt{v}_2$ are equivalent if

Algorithm 5 Statement synthesis algorithm.

procedure GENSTMTS($\phi, exprs$)
  Let $vars = \{v_1, v_2, \cdots, v_n\}$ be the set of variables that are primed in ϕ.
  Define $mod : vars \to 2^{exprs}$
  for all $v \in vars$ **do**
    $assigns \leftarrow \{v = e | e \in exprs, e$'s static type is a subtype of v's$\}$
    $calls \leftarrow \{e \in exprs | e$ is a method call, v is a subexpression of $e\}$
    $mod(v) \leftarrow assigns \cup calls$
  end for
  return $\{e_1; e_2; \cdots e_n; | e_i \in mod(v_i)\}$
end procedure

- both are primitives and `v1 == v2`,
- both are objects of the same type and `v1 == null ? v2 == null : v1.equals(v2)`, or
- both are arrays of the same type with the same number of elements and all corresponding elements are equivalent.

As an optimization, we call the `hashCode` method to speed up checking equivalence between objects.

We note that this definition depends on the `equals` method (and the optimization on the `hashCode` method) and so could potentially be unsound and incomplete, but this is unlikely to occur in practice; the Java documentation even states that the `equals` method defines an equivalence relation and that calling `hashCode` on objects that are equal with respect to the `equals` method "must" produce the same result [6]. We have not encountered any such problems during development.

In Section 3.2.1 we explained our deterministic synthesis algorithm in the context of synthesizing expressions but not statements. Our current implementation of `CodeHint` generates such expressions and can optionally assign them to a variable that the user marks.

We have developed heuristics for synthesizing more complex statements. Given a pdspec with n primed variables, we can synthesize a block containing n statements, each of which is either an assignment to one of the n primed variables or a call that passes one of the n primed variables as an argument. Algorithm 5 presents the details.

Discussion. We cache some information about the results of the expressions we explore (such as those that crash) in order to speed up subsequent searches. This makes it easier for users to try multiple different pdspecs, which can make it easier to find a desirable one. In addition, we display each expression that satisfies the user's pdspec as soon as we discover it, which means that users can often find their desired expression even before our search is complete.

Due to erasure, the type of generic methods and objects is not available at runtime [16]. We thus might generate statements that do not compile (e.g., if an argument to a method was erased), so we currently detect and filter out such cases. In addition, our current implementation might miss expressions that use the erased return value of a method (e.g., we might think a method returns an `Object` when it really is statically known to return a `String`), but we handle this case by downcasting the result to the actual type. We are investigating using reflection to get more precise type information. This ability to handle generics with runtime type checks highlights the fact that our approach can reason about generics without significant modeling effort.

The number of expressions that satisfy a pdspec can be quite large, especially when there are many candidates at higher depths. Since we present all of these expressions to the user, such a case could easily be overwhelming. We have, however, found that sorting the candidates by their likelihood usually puts the desired result near the top of the list. The ability to filter and sort the results can also be useful when there are many candidates. and we believe we can improve further by directly encoding the equivalence classes instead of eagerly expanding them.

Our search only uses the simple numeric constants `0` and `1` (and only uses them in infix expressions). Together, these two constants account for over two-thirds of the numeric constants used in practice and all other constants are used only rarely (see Section 4.2). We thus decided that including only these constants allowed us to make the synthesis more powerful without making the search space too much larger.

We note that our pdspecs allow users to express a form of duck typing. For example, the pdspec `x′.getData() != null` might lead us to find multiple results of incomparable types that contain a `getData` method.

4.1.1 Handling Side Effects

Our implementation detects in-memory side effects while evaluating expressions by installing watchpoints to listen for changes to fields. This allows us to log all the side effects of an evaluation and afterwards undo them and show them to the user. We use Java's security manager to disable external side effects such as deleting files and we block unknown native calls (which do not go through the security manager). Users can selectively enable or disable these features to control the tradeoff between efficiency, soundness, and completeness.

The basic idea is to use JVM watchpoints to break on field writes. For example, in the code given in Figure 4.4, if `this.mutable++` is executed, the watchpoint that waits for modifications to the `mutable` field triggers, and we can log the update on the `this` object with the old and new values. However, there are a number of complications.

While the JVM can notify us of field accesses and modifications (e.g., `this.mutable++`), it cannot notify us of array accesses and updates (e.g., `this.arr[0]++`). To work around this, for all array fields, we add access watchpoints and conservatively copy the array when it is first read. This copy includes all levels of nested arrays, but not deep copies of objects inside them. At the end of the execution, we can compare the current value of the array to

```
1 public class SideEffectsExample {
2     private int mutable = 0;
3     private final int immutable = 0;
4     private int[] arr = new int[] { 0 };
5     private Object unknown = null;
6     public static void main(String[] args) {
7         //Helper h = new Helper();
8         (new SideEffectsExample()).foo();
9     }
10    public void foo() {
11        System.out.println("Current location");
12    }
13 }
14 class Helper {
15    private int helperMutable = 0;
16    private static int helperStatic = 0;
17 }
```

Figure 4.4: Java code with a number of fields.

the stored value and reset it if necessary. If an array is passed as an argument to a method, we conservatively do this backup automatically.

As an extra implementation complication, in Java, arrays are subtypes of `Object`. So fields of static type `Object` (such as `unknown` in the example) can hold both normal objects and arrays. We thus have to be sure not to double-count changes to a field if it switches between being an object and an array during execution.

In addition, changing the value of a field through reflection does not trigger watchpoints on that field. We handle this case by adding breakpoints to the `Field.set` methods that do these changes and logging the change as if it were an actual field update. We similarly log calls to reading the value of a field array.

We would thus ideally put watchpoints on all fields, listening for modifications of non-final fields (e.g., we do not need a watchpoint on the `immutable` field) and accesses of fields that could be arrays (e.g., `unknown`) in the project. However, Eclipse performs poorly if you ask it to install this many breakpoints, so we need to reduce the number of breakpoints we install.

We currently install breakpoints only on fields of classes that are currently loaded, since while the JDK is very large, much of it will be unused in most executions. We then ask the JVM to notify us when classes are loaded and add new breakpoints for their fields. In fact, in such cases we only need to add breakpoints for their static fields, since by definition no instances of these objects existed before the execution, so we do not care about changes to

their instance fields. (However, we must be careful not to log and revert static initializations of static fields loaded during execution.) As an example, the `Helper` class has not been loaded, but future code could load it and change its `helperStatic` field. For similar reasons, as an extra optimization we ignore instance fields of classes that are loaded but have no instances in scope at the beginning of execution.

Furthermore, we need only install breakpoints on fields that are reachable from the current program location. We thus do a depth-first search from the local variables and static fields currently visible and do not install breakpoints on fields whose classes have been loaded but are not reachable. In our example, the fields of `Helper` are not reachable even if the commented line is uncommented and thus its `helperMutable` field does not need a watchpoint.

To handle effects outside the JVM, we disable external effects like writing files and sending network packets using Java's security manager. The JVM of course cannot do anything about effects in native calls, but we already let the user disallow those, in which case they are not a problem.

We apply a few more optimizations to improve this process. We ignore changes to fields of objects that were allocated after the execution began. We coalesce all writes to the same field of a given object and store only the initial and current values (unless they are equal, in which case we log nothing). We also ignore changes to certain fields that are harmless, such as the `hash` field of the `java.lang.String` class, which acts as a cache for the result of the `String`'s `hashCode` method.

To further reduce the number of breakpoints installed during this process, we run an offline analysis, shown in Algorithm 6, that determines fields that are never modified and do not install breakpoints on them. This is a simple syntax-directed analysis that computes fields that are never assigned, incremented, or decremented in the code. Note that the code contains special cases to handle arrays, since as mentioned above modifying an element of an array counts as modifying the array. It also avoids reporting final fields of arrays, since our implementation already ignores those. In our experiments, this analysis reduces the number of breakpoints by approximately 20%.

4.2 Probabilistic models

We built the probabilistic models described in Section 3.2 by analyzing the following projects:

- Tomcat 7.0.30 (a web server)
- FindBugs 2.0.1 (a static analysis tool)
- Hadoop 1.0.4 (a framework for distributed computing with a distributed file system)
- Hibernate 4.1.7 (a library that maps objects into a database)
- NetBeans 7.2.1 (an IDE)

Algorithm 6 Determining fields that are never modified.

Initialize $mod, maybemod, visible, simple$ with $\emptyset$.
for all field accesses e **do**
  if GETPARENT(e) is the target of an assignment, increment, or decrement **then**
    $mod \leftarrow mod \cup \{e\}$
  else if GETPARENT(e) is the argument to a method and e can be an array **then**
    $maybeMod \leftarrow maybeMod \cup \{e\}$ ▷ The callee might modify an array element.
  end if
  if e's declarer is final or not public or e is private or package protected **then**
    $invisible \leftarrow invisible \cup \{e\}$ ▷ Ignore externally visible fields that users can change.
  end if
  if e cannot be an array and e is declared final **then**
    $simple \leftarrow simple \cup \{e\}$ ▷ Ignore fields that can obviously not be modified.
  end if
end for
$notModified \leftarrow (allFields \setminus (mod \cup maybeMod)) \cap invisible$
return $notModified \setminus simple$
procedure GETPARENT(e)
  if e is the left-hand side of an array access **then**
    return GETPARENT(e's parent)
  else if e is the child of a parenthetical, a cast, or is the body of a conditional **then**
    return GETPARENT(e's parent)
  else
    return e
  end if
end procedure

- IntelliJ 123.72 (an IDE)
- Eclipse 4.2.2 (an IDE)

Together, these projects contained over 10.8 million lines of code.

In addition, we have explored a number of possible ways to improve these probabilistic models. While they greatly reduce the search space, they are fairly simplistic, so building an improved model would enable `CodeHint` to search for code snippets far more effectively. In addition, the probabilistic models also guide `CodeHint` toward snippets that are more likely to be the ones that users want. Improving them would thus make it more likely that we find code that users like instead of code that happens to satisfy their pdspec through luck. We also use the models to sort the results we show to the user, so as they improve our ranking heuristic does as well.

We have implemented a number of different probabilistic models and evaluated them to attempt to learn which will benefit `CodeHint` the most. Our results suggest that our current

models are effective but that we might be able to improve them, perhaps by combining multiple models.

Beyond unigrams. An n-gram model is a language model that predicts the next element given the previous $n-1$ items. A 1-gram, which is called a unigram, thus ignores correlations between multiple elements. Higher-order models, such as a 2-gram (bigram) or 3-gram (trigram), begin to take this information into account.

Our current model of the probability of a method or field access is a unigram model, as we assume that all subexpressions are independent. We compute the probability of calling each method and accessing each field, and the probability of a larger expression is simply the product of the probabilities of its subexpressions.

One example of how this independence assumption can fail deals with getting and setting the sizes of graphical elements. Users are probably more likely to set the size of newly-created objects and get the size of existing objects. Thus after calling `new JLabel("Hi")` to create a new graphical label, we are more likely to call `setSize` and less likely to call `getSize` than our current model would assume.

We note that our model that computes where constants are used is a form of a bigram model. For example, it knows that the integer `SwingConstants.LEFT` is more likely to be used in certain places in the Swing GUI library than as an index into a `String`. This partial bigram feature can significantly reduce the search space, suggesting that further similar improvements might greatly help as well.

Previous research [17, 43] has examined using such probabilistic models to improve autocomplete. We are solving a harder problem, but we hope to leverage some of this work.

Beyond methods and fields. Our current models only consider method calls and field accesses. They do not consider other language constructs such as binary and unary operators or control flow constructs. This has sufficed since we mainly target API code that uses mostly method calls, but we would still like to improve the models to cover the whole Java language. This knowledge would help us broaden `CodeHint` to be more useful for code that does not involve unfamiliar APIs.

We have already implemented some of this in the form model used in the stochastic synthesis algorithm described in Section 3.2.2. Table 4.1 presents its specifics. In addition, the probability of a static access (method call or field access) is 25.2% while an instance access has probability 74.8%. 45.1% of numeric constants are `0`, 22.5% are `1`, 5.3% are `2`, and 2.1% are `3`, followed by a long tail.

Exploring different models. We have implemented the following probabilistic models for method calls.

- The *char* models simply look at the characters in the source code. We have implemented a *unigram*, *bigram*, and *trigram* version of this model.

Form	Prob
Call	40.5%
Infix	14.2%
Field	8.1%
`null`	6.3%
`new`	5.2%
`this`	3.0%
Prefix	2.0 %
Array access	1.2%

Infix op	Prob
`==`	22.8%
`!=`	20.3%
`+`	19.5%
`&&`	10.6%
`<`	5.8%
`||`	5.7%
`-`	4.6%
`>`	3.7%
`>=`	1.6%
`&`	1.3%
`*`	1.0%

Prefix op	Prob
`!`	72.8%
`-`	22.0%
`++`	3.0%
`~`	1.1%
`-`	1.0%

Table 4.1: The probabilities used by our form model. Probabilities below 1% are omitted, as are certain things not in the grammar that we synthesize (Figure 4.2) such as postfix expressions and primitive constants.

- The token models break the source code into tokens. We have implemented a *unigram*, *bigram*, and *trigram* version of this model.
- The *calls unigram* model is a unigram model based on the name of the method: it simply stores how often methods with a certain name are called. Note that this model is a subset of the unigram token model, which contains this information as well as usage information about other tokens such as `if` and `+`.
- The *similar calls* model computes the number of times two methods are called in the same method. The probability of a certain method $\mathtt{m}_1$ being called from method $\mathtt{m}_2$ is then the average of the probability of $\mathtt{m}_1$ being called with every other method called in $\mathtt{m}_2$.
- The *bigram calls* model is a bigram model based on the names of methods: it stores how likely a method is to be called given the previous method called. Note that this model is different from the bigram token model, as in most cases the token that precedes a call is a period.
- The *calls type unigram* model is a unigram model based on the fully qualified name of the method, i.e., the fully qualified name of the type that declares the method combined with the name and signature of the method. It thus differs from the calls unigram and unigram token models because they cannot differentiate between two unrelated methods with the same name.

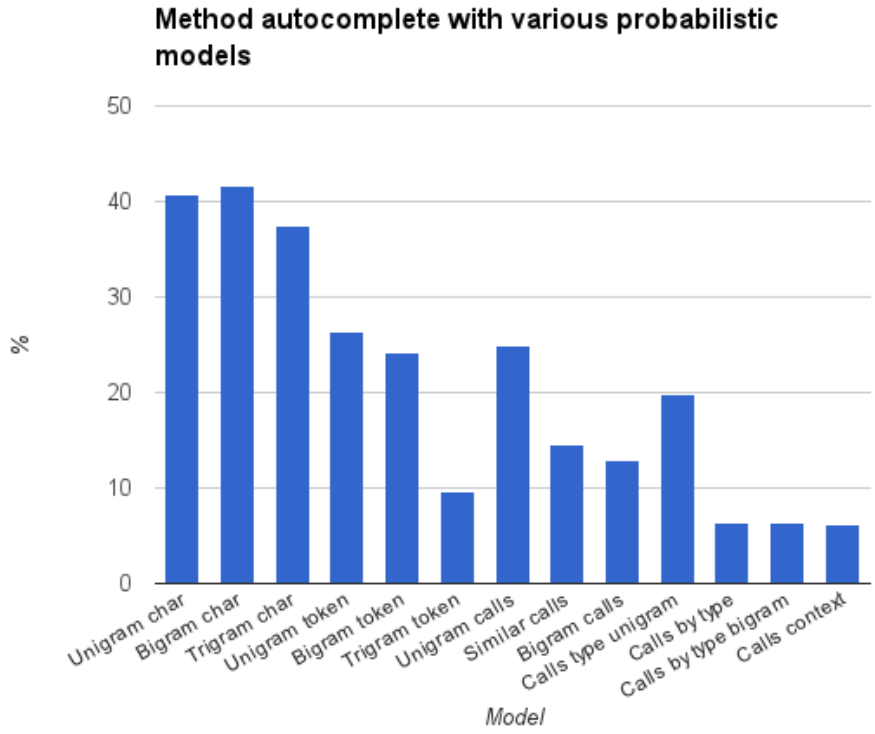


Figure 4.5: The median percentage ranking assigned to the correct method by various probabilistic models.

- The *calls by type* model is part of the model from Section 3.2 that computes the probability of a method separately for each receiver type. It computes the probability of a method call as the probability that a method of that name is called given that a method is called on something of the receiver's type.
- The *calls by type bigram* model is an extension of the calls by type unigram model that multiplies its probability by the probability that a method is called on a certain type.
- The *calls context* model is a generalization of a bigram that also considers the context in which a method is called. For example, it differentiates between using a call as the first and the second argument to another call.

We smoothed all of the models using the *one-count* method [20], which interpolates n-gram models with their $(n-1)$-gram counterparts. For example, it uses the unigram token model to help smooth the bigram token model.

We compared these models by using them to attempt to autocomplete methods. For each method call in our corpus (defined above), we used the type of the receiver to discover all the methods that could be called at that location. We then used each model to rank these methods and computed the percentile of the ranking of the method that was actually

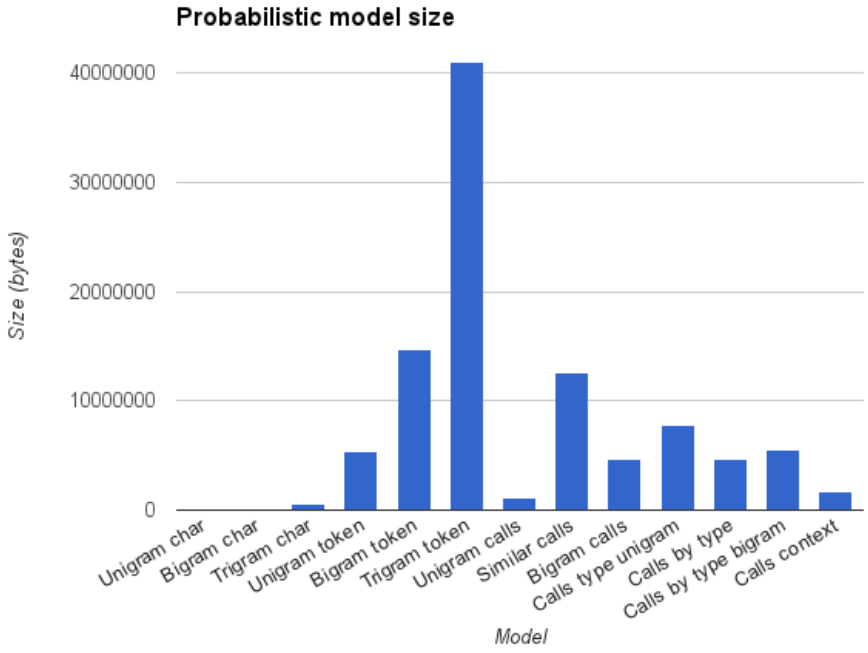


Figure 4.6: The sizes of various probabilistic models.

called. If the model assigns multiple methods the same probability as the correct one, we assume the correct method is assigned the middle ranking.

Figure 4.5 presents the results of this analysis. The calls by type, calls by type bigram, and calls context models are the best, with the correct method being near the 6% location, followed by the trigram token model at almost 10%.

Figure 4.6 shows the sizes of each of these models. We computed the sizes by serializing our unoptimized implementation of each model to disk and then compressing it. While the exact numbers are thus relatively unimportant, the differences between various models is interesting. For example, the similar calls and bigram calls models are approximately equally good at ranking methods but the former takes up approximately three times as much space.

These results suggest that our current models, which are similar to the calls by type, calls by type bigram, and calls context models, are very good.

4.3 Evaluation

We now show, through empirical analysis and two user studies, that `CodeHint` is sufficiently scalable and that it makes users more productive.

	Normal algorithm						Side effects		Brute force
	Depth 2		Depth 3		Depth 4		Depth 2	Depth 3	Depth 3
	#	Time	#	Time	#	Time	Time	Time	#
P 1	34	0.1	611	0.6	19644	6.1	0.1	1.5	3397815
P 2	52	0.1	727	0.7	34763	8.3	0.3	2.5	948871
P 3	53	0.1	1091	0.6	125217	17.8	0.3	4.3	6728128
P 4	7	0.1	53	0.2	583	0.9	0.1	0.1	135
P 5	22	0.1	239	0.3	2979	1.0	0.1	0.4	147437
S 1	8	0.2	223	1.0	1402	2.2	0.3	2.4	39439
S 2	12	0.1	275	1.0	2044	2.5	0.3	2.6	51080
S 3	70	1.0	814	1.5	6652	6.6	0.7	8.3	1867197
S 4	103	0.3	842	1.5	22144	9.1	0.3	7.7	61239025
S 5	32	0.2	599	1.4	35843	12.6	0.5	11.0	270911
R 1	20	0.1	93	0.2	893	0.6	0.2	0.8	36264
R 2	12	0.0	137	0.2	1161	0.5	0.0	1.0	1080
R 3	8	0.2	20	0.2	58	0.7	0.2	0.7	116
R 4	6	0.0	19	0.1	68	0.3	0.1	0.4	48
R 5	24	0.2	226	0.4	1998	2.3	0.2	24.9	103124
Avg	30.9	0.2	397.9	0.7	17029.9	4.8	0.3	4.6	4988711.3
Med	22	0.1	239	0.6	2044	2.3	0.2	2.4	103124

Table 4.2: An empirical analysis of our algorithm. Each row represents one task from the first user study. The first six result columns show how our algorithm performs at different depths, the next two show the performance when we undo side effects, and the last shows the performance with key parts of the algorithm disabled. The # columns show the number of expressions searched and times are in seconds.

4.3.1 Empirical Evaluation

Scalability. To analyze the efficiency of our implementation and demonstrate its scalability with regards to depth (as defined in Section 3.2), we ran the tasks used in our first user study (described in Section 4.3.2) with a typical pdspec (i.e., the one most frequently used by subjects in the study) and a skeleton indicating that we should not search constructors and operators. For each task, we varied the maximum depth searched, and for each depth, we recorded the number of unique expressions evaluated and the total time taken.

The experiments were performed on an Intel Core 2 Duo E6850 with two 3 GHz processors and 3 GB RAM (although our current implementation is single-threaded) running Linux 3.15.8, Java 1.8.0_11, and Eclipse 4.4.0.

The results in Table 4.2 show that our current implementation, when not undoing side effects, can explore a search space of depths 1, 2, and 3 in well under one second and depth 4 in under five seconds. We do not show depth 5, for which we have not optimized and all but six benchmarks timeout after two minutes. When undoing side effects, our implementation takes well under one second to search depth 2 and under five seconds to search depth 3. We believe these results are more than sufficient for a useful tool, and results from the user studies show that users agree.

As part of our methodology, `CodeHint` finds all expressions in its search space that satisfy the user's pdspec, not just one. Since we display each such candidate expression as soon as we discover it, users can usually see some candidate expressions well before the search terminates.

To measure the effectiveness of various parts of our algorithm, we searched depth 3 with pruning, equivalence classes, and the probabilistic model disabled. The results, given in the rightmost column of Table 4.2, show that these improvements significantly reduce the search space, enabling `CodeHint` to find larger expressions. Without them, it would not be able to search even depth 3.

To show that `CodeHint` can synthesize large code fragments, we tracked the size of the expressions explored for each task. The largest such code snippet contained ten method calls, showing that our algorithms can indeed scale to non-trivial code fragments.

Analysis of expressions. To justify our algorithm, we analyzed five medium to large open-source Java projects. The key result is that the vast majority (99%) of assignment statements are actually quite simple (with a depth of at most 4).

We analyzed four of the projects presented in Section 4.2 (Hadoop, Tomcat, FindBugs, and Hibernate) as well as JDK, the implementation of Java.

For each program, we analyzed the right-hand side of assignment statements, which is the exact class of statements we currently generate. Hadoop version 1.0.4 contains 47,248 such expressions, Tomcat 7.0.30 contains 30,740, FindBugs 2.0.1 contains 18,589, Hibernate 4.1.7 contains 54,580, and JDK 6.25 contains 227,731.

We began by analyzing the depth of the expressions. All five programs followed a similar pattern: on average, 27% of expressions had depth 1, 52% had depth 2, 15% had depth 3, 5% had depth 4, and 1% were more complicated. This shows that the vast majority (99% in our study) of statements in real Java code have depth at most 4, which our results above show that `CodeHint` can easily search.

To show that these results also hold for code programmers struggle to write, we repeated the same experiment on code snippets gathered from questions asked on the popular Stack Overflow website. We examined 43 code samples containing 3333 lines of code and found nearly the same distribution: 25% had depth 1, 59% had depth 2, 13% had depth 3, and 3% had depth 4.

Figure 4.7 shows these results. The numbers are very similar across all the projects, which suggests that they accurately represent reality. While users might want to synthesize

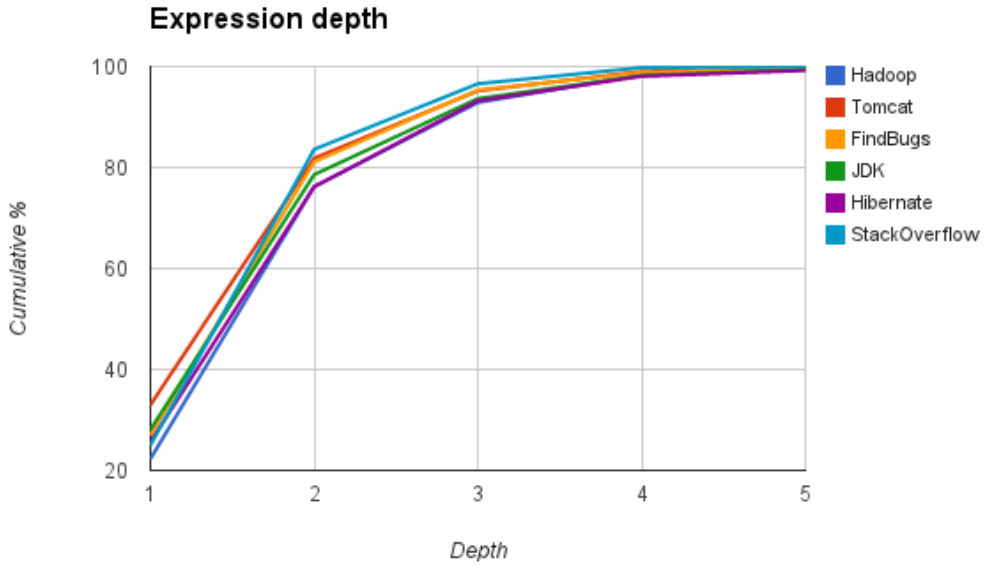


Figure 4.7: The cumulative percentage of expressions with at most the given depth.

multiple lines of code at a time, we believe these results suggest that our current algorithms can help real programmers.

We conservatively bounded the number of cases in which we pruned calls to methods that could be called with many different arguments by analyzing the number of calls with a given number of arguments. All five programs followed a similar pattern: 51% of calls had no arguments (excluding the receiver), 31% had one argument, 12% had two arguments, 3% had three arguments, and 3% had four or more. Thus 94% of calls in practice contain two or fewer arguments and hence little combinatorial explosion, and so biasing our search to avoid the rest seems beneficial. We additionally note that just because a method has multiple arguments does not mean that there will be many ways to call it, as arguments are often of types that have few values in practice, such as singletons. Our heuristic will not avoid such calls.

4.3.2 User Studies

We conducted two user studies to evaluate the usability of `CodeHint` and learn how developers use it. The first study focused on constrained single line code edits and the second focused on larger open-ended tasks and used an improved version of `CodeHint`. Observations from these studies generated additional areas for improvement that we implemented in the meantime, including the skeletons of Section 3.2.

Methodology

Study 1: Line-level tasks. For the first study, we created three scenarios with five subtasks each that mimic code completion tasks programmers face in practice. To focus on particular tasks where `CodeHint` could be applied, we provided working wrapper code that participants had to extend. The Parse (or P) scenario manipulated strings and parsed email headers and command-line arguments. The Swing (or S) scenario created a small GUI. The RandomWriter (or R) scenario created a Markov model to generate output that looked similar to input text.[2] We chose the first two scenarios to represent common tasks involving APIs and the third as an example application. The first part of Section 2.1.1 is a slightly modified version of one of the tasks. Appendix A lists the code for these scenarios.

At the time of the study, `CodeHint` could solve thirteen of the fifteen tasks; we included the remaining tasks to see how subjects handled cases it could not solve (both could have been found with a slightly higher search depth and can be found by the current version of the tool).

In our within-subjects study, each user received a random assignment of *control*, *experimental*, or *choice* conditions to scenarios. In the control condition, users could not use `CodeHint`, in the experimental condition they were required to use it (although they could write code normally if it failed), and in the choice condition they could decide whether or not to use `CodeHint` for each task. We counterbalanced the order of the experimental and control groups and assigned the choice condition last so participants had experience both using `CodeHint` and writing code normally.

Study 2: Open-ended tasks. The tasks in the second study were larger – each task required writing three to sixteen lines of code that used complex APIs. Two scenarios had three tasks each, again scaffolded by wrapper code. We also included comprehensive tests to ensure that subjects wrote correct code. The Eclipse scenario implemented a simple Eclipse profiler plugin and the Note scenario involved writing a GUI note-taking application that synchronized data to the cloud. Example tasks included finding all the objects in the heap of a program running in Eclipse and adding a menu item that made selected text bold. Appendix A lists the code for the Eclipse scenario.

Each participant solved one scenario in the control group and the other in the choice group (as defined above). Subjects in the choice group could solve tasks using a combination of `CodeHint` and traditional techniques. Both the scenarios and the groups were ordered randomly. We additionally allowed subjects who could not solve Eclipse tasks in the control group to solve them with `CodeHint`. The tasks were designed to be difficult to solve, so we stopped subjects if they had not completed a task after twenty minutes.

[2]This was directly inspired by an assignment given in the Stanford introductory programming courses.

Participants

Nine subjects initially completed the first study; another five were recruited later to complete only the choice condition of the same study with an improved version of the tool to collect additional data. Twelve were graduate students in Computer Science at UC Berkeley and two were undergraduates.

Another fourteen subjects completed the second study, which used a further improved version of `CodeHint`. Ten were undergraduates and four were graduate students in Computer Science at UC Berkeley.

In both studies, participants practiced on some training tasks first and completed a post-test questionnaire. They were allowed to use a web browser to search for help. None of the subjects had ever used `CodeHint` before, but all were somewhat familiar with both Java and Eclipse.

Measures

We defined the following measures:

- *Task completion time*: Time taken to either complete or abandon a task.
- *Task completion rate*: Percentage of tasks users successfully completed.
- *Code quality*: Number of bugs in participants' task code.
- *Tool choice*: Fraction of tasks in the choice condition for which participants opted to use `CodeHint`.

Results

We first discuss quantitative results followed by qualitative impressions of how our participants used `CodeHint`.

Productivity and preference. *Completion time:* In all but one case, participants in the first study completed all tasks, so we focus our analysis on task completion times. On average, subjects using the improved version of `CodeHint` completed tasks in 46 seconds and control participants took 97 seconds (see Figure 4.8). This difference is significant (two-sample $t(11) = 2.42, p = 0.033$, two-tailed). This suggests that programmers are more productive when using `CodeHint`.

Completion rate: Participants in the second study did not complete many tasks, so we focus on task completion rates. Figure 4.9 shows the task completion rate for users in our second study with and without `CodeHint`. On average, subjects using `CodeHint` completed 69% of sub-tasks while those not using it completed 27% ($\chi^2(1, N = 14) = 26.06, p < 0.001$), which strongly suggests that programmers complete more difficult API tasks when using `CodeHint`.

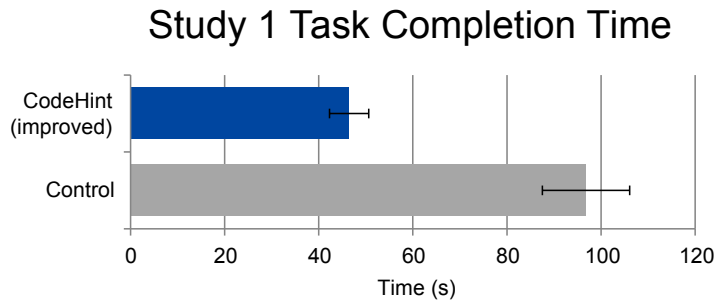


Figure 4.8: The task completion time of subjects in our first user study. The error bars show the standard error.

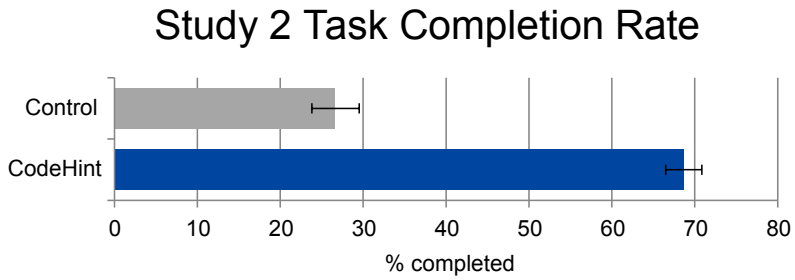


Figure 4.9: The task completion rate of subjects in our second user study. The error bars show the standard error.

Code quality: Participants introduced 11 bugs in 122 tasks solved with `CodeHint` in our first study and 24 bugs in 93 tasks in code written without it (two-sample $t(12) = 2.81, p = 0.015$, two-tailed). This suggests that `CodeHint` improves code quality. To focus on the completion rate, we gave subjects in the second study a comprehensive test suite, so they wrote almost no bugs.

Tool choice: In the choice condition, each user could elect whether or not to use `CodeHint` for each task. On average over both studies subjects used `CodeHint` 71% of the time, suggesting that users found `CodeHint` valuable.

In the questionnaire, participants in both studies rated the overall usefulness of `CodeHint` at an average of 7.7 out of 10 (with a standard deviation of 1.2). All users reported that they would use `CodeHint` for their own development if it were available for their language and editor and had some simple flaws fixed. Six of the subjects asked for the plugin shortly after completing the user studies and installed it. When describing the tool, users stated that "[Using the tool] seems natural" and was "way better" for some tasks than the traditional approach.

Qualitative results. From manual inspection, it seems that users chose not to use `CodeHint` mainly for simple statements that they already knew how to write, often by using autocomplete. They seemed to use it when it was easy to give a pdspec or when the code involved using an unfamiliar API.

When watching the subjects use `CodeHint`, we noticed some interesting patterns. Some users chose an expression without carefully examining the list of candidates, which sometimes caused them to choose an incorrect expression. On the other hand, examining the entire list of candidates and considering the difference between related expressions allowed some users to realize that they had written bugs in similar code earlier.

`CodeHint` presented only a small number of candidates to the user after the initial pdspec: in the first study, the average number of candidates across all episodes was 13 and the median was 2, and in the second, the average was 34 and the median was 3. When refining an existing set of candidates, users in both studies provided pdspecs that reduced the number of candidates by 31% on average. However, 47 out of the 89 refinements did not reduce the size of the candidate set at all, mostly because all the candidates were already equivalent on all possible inputs. Ignoring those, the average reduction was 66%.

Choosing a pdspec requires trading the strength of the specification for the ease of encoding it and the cost of evaluating it. To examine this tradeoff, we classified all of the pdspecs used by subjects while completing the user studies and found that in the first study, 53% demonstrated the desired value, 33% gave the desired type, and 14% were arbitrary predicates. In the second study, subjects used only type demonstrations for the Eclipse scenario (as its difficulty lay in finding and using complex types), but for the Note scenario they gave 51% value demonstrations, 38% type demonstrations, and 10% arbitrary predicates. This shows that users can get benefits with `CodeHint` even while demonstrating simple pdspecs but that the ability to provide more expressive pdspecs is sometimes valuable.

We list the arbitrary pdspecs we saw below, omitting duplicates, to give a flavor of the pdspecs users found helpful.

```
1 x′.contains("-x") && x′.contains("-y")
2 x′.get(0).equals("-x") && x′.get(1).equals("-y")
3 x′.getWidth() == w && x′.getHeight() == h
4 x′ >= 0 && x′ < 3
5 x′ >= 0 && x′ < followers.size()
6 x′.toString().contains("home")
7 x′.charAt(0) == '<'
```

The first two pdspecs express the same property, but the second is stronger than the first as it contains ordering information. Similarly, the fourth and fifth pdspecs also express the same property (in this case, trying to find a random integer in a range), but the fourth uses the concrete value of `followers.size()` and so is more context-dependent. The sixth pdspec differentiates between many values of the same type by finding the one whose `toString` method returns a desired value), which allowed the user who wrote it to avoid figuring out how to express a property about a complicated API (and directly inspired Example 2.1.1 in

Section 2.1).

When in the experimental group, one subject manually created a new test case that made giving a pdspec significantly easier. This highlights the importance of the test case used; the same pdspec can yield significantly different results for different test cases.

During the first user study, several users found interesting ways to synthesize complicated expressions. We had included two tasks that were deliberately outside the scope of what was then directly solvable with a single search in the tool. Two users, once they realized this, split the task into two sub-tasks by adding a temporary variable. They computed the temporary using the tool and then gave their original pdspec again, letting the new temporary be used in the generated expressions. Another user gave a new pdspec that led to a related (but simpler) expression and then modified that by hand to find the correct expression. These examples show that `CodeHint` can be useful even when it cannot find the desired expression.

4.4 `CodeHint.js`: Bringing `CodeHint` to JavaScript

Thanks to the rapid growth of the web, JavaScript has become one of the most popular programming languages [8]. Thanks to recent advances in Just-in-Time compilers for JavaScript [29], it has become feasible to write large projects in JavaScript and even automatically compile native code to JavaScript [9, 97]. JavaScript is thus likely to become even more popular in the future.

Unfortunately, due to its dynamic nature, IDE support for JavaScript is far behind that for languages such as Java [27]. Even simpler tasks such as code navigation and completion often do not work well, and there are few advanced analyses.

We thus believe that a tool such as `CodeHint` is uniquely placed to benefit JavaScript programmers. Due to its dynamic nature, it can easily discover information, such as the dynamic types of variables, that is not available to most static IDE tools. Thanks to the rising popularity of JavaScript, such a tool has the potential to have a large impact on programmers.

We have implemented a prototype tool called `CodeHint.js`. Its source code and a demo (shown in Figure 4.10) are available at https://github.com/jgalenson/codehint.js. This prototype can currently synthesize code involving arithmetic, array and property accesses, and function and method calls and uses equivalence classes to reduce the search space, but it does not correctly handle side effects.

Beyond requiring a re-implementation in a different language, many of the main concepts of `CodeHint.js` are different from `CodeHint`.

Dynamic features. Language features such as dynamic typing, variadic functions, and undefined values will require changes to the search algorithm `CodeHint` uses. These features are used in a non-trivial fraction of real JavaScript code [72].

While JavaScript's dynamic type system provides an opportunity for `CodeHint.js` to benefit developers, it makes the search space much larger.

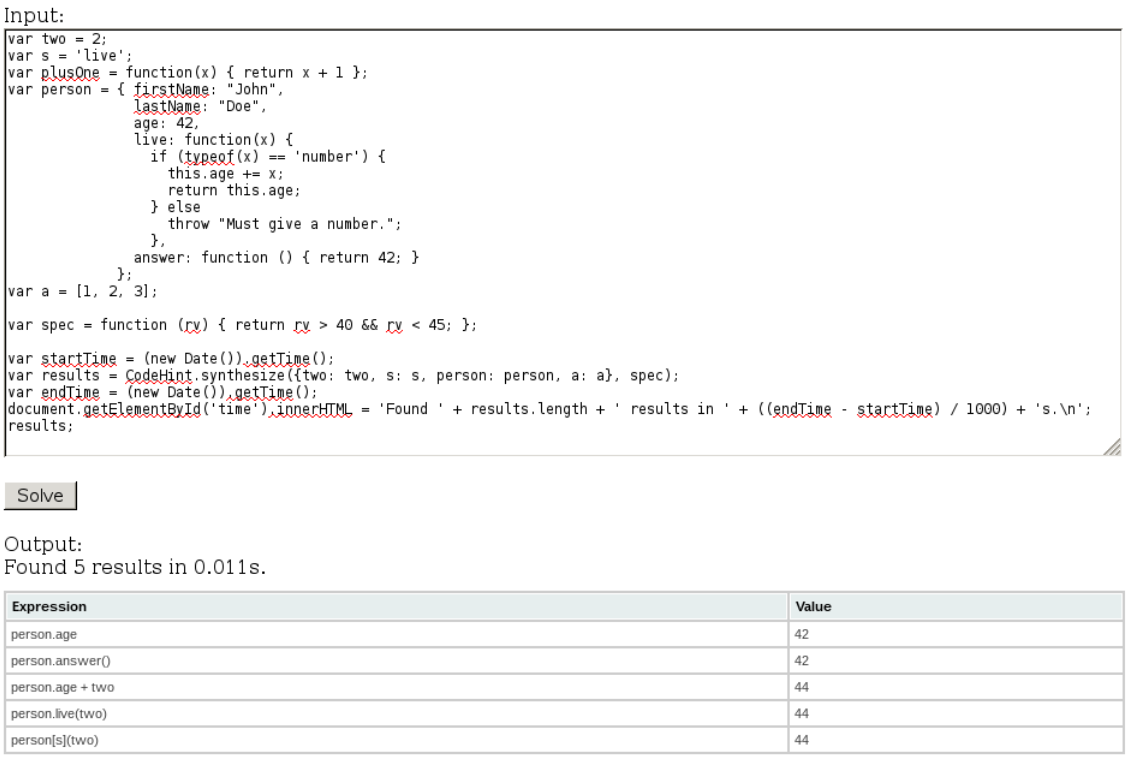


Figure 4.10: A simple web interface to the `CodeHint.js` prototype.

For example, consider the following Java code.

```
class Foo {
  String bar(File f, int x) { ... }
}
```

During our search, when we encounter an object of dynamic type `Foo`, we will try to call all its methods, including `bar`. Thanks to Java's static type system, we will know that `bar` expects two arguments, a `File` and an `int`, and so we will call it with only arguments of the correct types.

Now consider the corresponding JavaScript code:

```
Foo.prototype.bar = function (f, x) { ... };
```

As before, when we encounter a `Foo` object during a search, we will try to call its `bar` method. We will see that it expects two arguments but will not know what their types should be. A naïve search, which our prototype currently implements, would thus call the method with all possible arguments, which would greatly increase the search space.

A related difficulty is variadic functions. In the above example, the JavaScript `bar` method can be called with any number of arguments, not just the two that it declares. We might thus want to call it with all combinations of three arguments or even any subset of possible arguments, either of which would greatly increase the search space.

An additional problem is JavaScript's `undefined` value. This value can represent an unassigned variable or parameter, a missing field, a function that does not return a value, or more. It thus sometimes, but not always, represents an error, making it difficult for an automated search to decide whether it is useful.

Functional features. JavaScript supports first-class functions, unlike Java, and so functions can be passed as arguments and returned from functions. First-class functions greatly increase the search space, as `CodeHint.js` must now treat functions like other values such as integers and add them as candidates.

In addition, JavaScript supports closures and anonymous functions. Creating new anonymous functions as part of our search would greatly increase the search space, as each expression we would normally search could be contained inside an anonymous function with any number of arguments.[3]

Usability. `CodeHint` was developed as a plugin to Eclipse, which provided many of the features needed to implement the tool (such as finding the local variables and dynamically evaluating code). Since JavaScript contains more dynamic features than Java and developers use many editors when programming it, we have decided to implement `CodeHint.js` as a library.

This allows our code to be portable. As an example, in our current prototype, using `CodeHint.js` is as simple as:

```
CodeHint.synthesize({ x: x, person: person },
  function(rv) { return typeof rv == 'string'; })
```

This code works in any environment, including Node.js or in a web browser.

This approach contains extra difficulties, however. Certain features available to Eclipse plugins such as `CodeHint` are not available to JavaScript code. For example, Eclipse provides a simple API to query for the local variables, which we use to seed our search, but JavaScript does not seem to have an API with the same functionality. We currently work around this with more verbose syntax; the first line of the code snippet above requires the user to provide values (and their strings) manually to seed the search.[4]

[3] Java's anonymous classes provide similar functionality, but they are less common so `CodeHint` does not search them. The newly-released Java 8 supports closures directly, but `CodeHint` does not yet support them.

[4] It is worth noting that while this syntax is more verbose, it is also more powerful, as it allows the user to direct the search space in a manner somewhat similar to `CodeHint`'s skeletons by providing complex expressions or not providing certain variables.

Chapter 5

Programming by Demonstration with Little User Interaction

We now show how we can use `CodeHint`'s algorithms to build a Programming by Demonstration tool. We begin in Section 5.1 by describing our overall approach and then describe the two key new algorithms in Sections 5.2 and 5.3. We then discuss our implementation in Section 5.4 and evaluate it in Section 5.5.

5.1 Approach

We now give more details of our approach, which is graphically depicted in Figure 1.1.

A trace consists of a sequence of actions given by the user. These actions are a superset of the statements in the host language because they allow heap values to be used as expressions. For example, `x = y`, `x = 1`, and `Node@123.x = 42` are all valid actions while `Node@123 = x` is not valid (where `Node@123` represents a heap object). This extension allows users to give demonstrations that refer to values in addition to names, and it allows graphical programming patterns such as dragging pointers and values to make assignments.

A partial program consists of a normal program in the host language with two additional types of expressions. *Unseens*, denoted by `?`, represent code that has not been encountered, e.g., on a branch that has not yet been taken, and *holes* like `?(x,y,z+1)` represent an ambiguous expression that could be any of the inner expressions. We note that these partial programs can sometimes be executed automatically, as they might not contain any holes or unseens or all the possible expressions in a hole might yield the same result.

We translate traces into partial programs by finding expressions that can result in the actions seen in a trace. For example, the action `Node@123.x = 42` might become the statement `?(p,q).x = ?(42,a)` if `a` has value `42` and `p` and `q` point to the object used in the trace. We use a slightly modification of the algorithms presented in Section 3.2; we discuss the details in Section 5.4.

We have developed pruning algorithms that automatically resolve many ambiguities in the code. We discuss them in Section 5.2.

Since we resolve some of the ambiguities in partial programs by asking users to give more traces, we need to be able to find inputs whose traces will help us. An input is helpful if it cannot be executed without encountering an ambiguity (in such a case the user will demonstrate the correct action and resolve the ambiguity). For example, given the statement `?(p,q).x = ?(42,a)`, an input is helpful if, when executed to this line, `p != q` or `a != 42`; if neither holds, then the user can give us no new information about this line. We discuss how our implementation finds helpful inputs in Section 5.4.

If the partial program is complete (i.e., has no holes or unseens), we will not be able to find any helpful inputs and so will terminate our algorithm and present the user with the final code. However, in some cases we might not be able to find a helpful input even when the partial program contains ambiguities. This may either be due to aliasing (e.g., we will never be able to disambiguate `?(x,y)` if `x` is always equal to `y` at that point) or to incompleteness in our search for useful inputs. In this case, we will present the user with a partial program that she must complete manually.

In the interactive trace process, we help the user give us a new trace. At each step of the program, we suggest the next action to take if there is no ambiguity in the current line and suggest the possible actions we expect the user to take otherwise. These suggestions make it easier for the user to give a trace, and her disambiguations help us complete the partial program.

Since a trace might not explore all branches in a conditional, we must handle control flow specially. During an interactive trace, a user may note that a conditional has been encountered and mark where the branch begins and ends. We then synthesize a `?` unseen in our partial program to note that we have not seen the other branch and should try to explore it later. However, this approach somewhat breaks our trace abstraction, as it forces the user to think about something not encountered (the untaken branch) on a single trace. We thus have extended our approach to automatically infer conditionals if users do not specify them (see Section 5.3 for details).

If the user marks that a loop is encountered, we continue normally until the user marks that the first iteration has completed. We then immediately synthesize a partial program for the loop body and prune it to resolve some ambiguities, which allows us to suggest actions to the user and resolve ambiguities as later iterations are encountered during the same interactive trace. In addition to reducing the burden on the user, this technique ensures that a user need only demonstrate each line of code once (although she might have to respond to a finite number of extra disambiguation queries). There is thus a finite bound on the amount of work a user must do during an interactive trace.

Input `a = [6, 8, 1, 3, 7]`

Trace:

```
if (|a| > 1)
  last = 7
```

Partial program:

```
if (|a| > ?(1, a[2]))
  last = ?(a[4], a[|a| - 1])
```

Figure 5.1: A contrived example of our pruning algorithm. The top shows a concrete input, the left shows the user's trace, and the right shows the generated partial program.

5.2 Pruning

After demonstrating each line of code once, the main work our approach requires of users is resolving ambiguities, so by doing this automatically we can greatly reduce the burden on the user. We now describe the techniques we use for this purpose.

Given a partial program and an input, we execute all paths through the partial program (which means executing all complete programs the partial program represents) and note which crash or fail the user-provided postcondition. Then, for all holes that were seen on all successful paths, we remove all the possibilities that were never used on a successful path.

An example is shown in Figure 5.1. The top part of the figure shows the input array, the left shows the user's trace, and the right shows a partial program we might synthesize that corresponds to this trace. Our algorithm then explores all four possible paths through this partial program on a different input. On the input array `[9, 2]`, three of these paths crash, and only one is successful. We can thus automatically prune the expressions (`a[2]` and `a[4]`) that were never used on a successful path.

Intuitively, each successful path could be the complete program desired by the user, so any expression in a hole that is used on such a path is potentially needed and must be kept. If we assume that one of these paths corresponds to the desired program, all expressions not used on any successful paths can be safely discarded.

We note that we cannot prune possibilities for holes that we do not see on all successful runs. For example, consider a program that contains a loop whose condition is a hole. Some paths might not enter the loop (e.g., if the program iterates over all elements of an array) while others will. If a potential expression inside a hole in the array always fails on such an input, we cannot be sure if this is because it is invalid or because the desired program simply does not execute the loop.

We consider all paths that encounter an unseen as being successful. This is the best we can do, as the code inside could do arbitrary computations and yield a correct output.

Some of the paths we explore might not terminate. For example, if the user demonstrates a loop with an induction variable `i` that is initially `0` and is updated to be `1`, we might generate the hole `i = ?(1, i + 1)` where the first possibility can easily lead to an infinite loop. Since it is impossible to always determine whether some code will halt, we must thus timeout long-running executions and consider them to be successful.[1]

[1] Our implementation actually treats timeouts as failing, which is unsound but works well in practice.

This algorithm is sound, as if the partial program contains a correct program that does not crash on any valid input, this algorithm will only remove possibilities that do not occur on any correct programs represented by the partial program. It is also as strong as possible (with respect to the inputs tested), as it only keeps potential expressions that might be used in the user's desired program.

Using strong postconditions can greatly improve the effectiveness of this approach. Even without any postcondition, it can still be effective at pruning expressions that cause crashes, but strong postconditions can allow it to remove many (sometimes even all) incorrect possibilities, as we show in Section 5.5.

We note that this pruning is more effective when there are fewer unseens, as we must consider all paths that contain one to be successful. This suggests that when generating inputs for interactive traces, we prioritize those that bring the user to unseens, which we currently do (see Section 5.4).

This algorithm explores all potential paths through the partial program (somewhat similar to model checking [21, 40]) and so is exponential in the size of the program. We thus apply a number of optimizations that make it quite efficient in practice.

Equivalence classes. We can separate the possibilities for a hole into equivalence classes, grouping together elements that evaluate to the same result. Paths then contain not single expressions for each hole but rather sets of expression, all of which yield the same result. If we execute the same hole more than once (due to loops or recursion), we split the current equivalence class into new classes each time its members are no longer equivalent. This optimization usually significantly decreases the search space.

Formally, we define two possibilities for a hole as equivalent on a trace if they yield the same result whenever that hole is encountered on the trace. There is clearly no benefit from treating such equivalent possibilities separately.

As an example, assume that we encounter the hole `?(x.right,y,x.left,null)` during a search. If `x.right == y` and `x.left == null`, we need only search two paths ([`x.right`,`y`] and [`x.left`,`null`]) instead of all four. If on this same path we return to this hole due to a loop and `x.right != y` but `x.left == null`, we must consider three possible paths ([`x.right`], [`y`], and [`x.left`,`null`]).

Shortcuts. We can avoid searching paths that we know will not help us. Since we can only prune possibilities from holes that were seen on all successful paths, once we have seen a successful path without a certain hole, it is only useful in giving us information about other holes. We call such holes *unhelpful*. Thus if a path uses, for each helpful hole, a possibility that we have seen on a successful path (or, when combined with the equivalence classes optimization, an equivalence class all of whose possibilities we have seen on a successful path), we do not need to explore different possibilities for the unhelpful holes at the end of the path. This optimization greatly reduces the search space when there are many unhelpful holes, which can occur due to conditionals, loops, or unseens.

As an example, take the partial program shown in Figure 5.1. For a given input, we might explore a path that does not enter the conditional, and so we know that the second hole is unhelpful. We might then explore a path that chooses `1` for the first hole and `a[4]` for the second and find that it is successful. We then do not need to explore the path that uses `1` for the first hole and `a[|a|-1]` for the second; if it were successful, we would want to mark `1` as being used on a successful path, which we have already done, and if it were unsuccessful, we would simply continue.

As soon as a path encounters an unseen, we immediately mark it as successful. Any holes not seen on this path, including those potentially reachable from the unseen, will thus be marked as unhelpful. This also helps us avoid searching paths we know will not give us new information.

Small inputs. When pruning multiple times in sequence, we can first prune with smaller inputs and then move on to larger ones. Since smaller inputs are more likely to be edge cases and not execute the entire program, this can allow us to quickly remove some possibilities, reducing the search space for subsequent prunes with larger inputs.

Fast pruning. Since this algorithm explores all paths through the partial program for a single input, it is best used multiple times for multiple different inputs. Despite these optimizations, it is still exploring a large search space and so may be slow.

We have thus developed a lightweight and fast version of this algorithm that we call the *fast* algorithm (as opposed to the *full* pruning discussed earlier). Given an input, we evaluate the partial program as far as we can, aborting when we encounter either an unseen or a hole whose possible expressions do not all evaluate to the same value. During this process, we prune any possibilities we encountered whose execution crashed.

This algorithm is much weaker than full pruning, as it can only execute the partial program until the correct path becomes ambiguous. However, it can usefully prune possibilities that crash on some inputs, and so works especially well when given edge cases. In addition, it is very fast, as it only partially executes a single program, so it can be applied many times to many different inputs. Fast pruning can be used after full pruning, as the pruning done by the latter might allow the former to explore more of the partial program. We note that this new algorithm only prunes possibilities that the full algorithm would also prune, as if this algorithm prunes a possibility, then all paths the full algorithm explores also crash on that possibility.

5.3 Lazy Control Flow Inference

Requiring that users notify us of control flow constructs by marking the beginning and end of all conditionals and loop iterations both increases the amount of work they must do and increases the cognitive load, as they must now think about more than just the current trace (e.g., by realizing that there is an untaken conditional at a certain point).

We would thus like to infer the control flow structure of the program. Following the approach given in Section 5.1 for synthesizing straight-line code seems impractical, however; if we assumed that any point in the program could have conditionals or loops with unknown guards and bodies, we would have many more holes and ask the user far too many questions. Thus rather than eagerly attempting to guess the control flow structure of the program, we try to infer it lazily.

To simplify the presentation, we begin by considering only conditionals and later extend the approach to cover loops.

With this lazy approach, we take the user at her word: we use any conditionals she gives us but assume that there are not any more. Thus our partial programs look exactly like they did before and users are not required to mark conditionals initially (although they may do so if they wish). Our assumption may turn out to be incorrect when in later traces the control flow diverges from earlier traces by going down a different branch. When this happens, we have learned that there is a conditional and seen both of its branches.

To help illustrate this approach, consider a user trying to synthesize a program that contains a conditional. The initial input might take only one path through that conditional, and it may not be immediately obvious to the user that there is another path (e.g., it might be an edge case). Thus in her initial demonstration, the user does not mark the beginning or the end of the conditional and does not show the condition. We then synthesize a partial program without a conditional. A later input might drive the program down the other path of the conditional. The user then recognizes that a different action is needed and will realize that there must be a conditional. She can demonstrate the condition (or let us synthesize it) and show what occurs in the new branch and when it ends. Our partial program then contains the conditional, and we continue.

The user might recognize that a new conditional is needed, but even if she does not, we can in certain cases infer that a conditional might exist. If during input generation or pruning we encounter an input for which the partial program contains no successful paths (including an input that can be executed fully but fails the postcondition), we know that a missing branch might be executed on this input. If the user tries to do an action that we do not expect during an interactive trace, she might be demonstrating a new branch of a conditional. If due to pruning or a user's demonstration we remove all the possibilities for a hole, that could be a sign that there is a missing conditional above the current point in the trace.

We will not always run into such an error; we might synthesize a complete program that happens to be missing a conditional. Unfortunately, we have no easy way to recognize this case. All we can do is generate many input/output pairs and check that they are valid. If this approach finds an error, we realize that we might have missed a conditional and proceed. Otherwise, if a conditional is missing, our final program will be incorrect. While this is unfortunate, if neither the programmer nor her correctness condition recognize the error, there is nothing more we can do.

When we recognize that the partial program is incomplete on a certain input, we ask the user for an interactive trace. When she realizes that there must be a conditional, tries to

demonstrate an action we do not expect, or if the current line would crash, we can recognize that there is an error at the current point. The error could be due not to a missing conditional but instead to our synthesis algorithm not finding the correct code or to mistakes in the user's demonstrations. We thus simply ask the user to fix the error, which might involve marking a conditional.

Joins. Once we recognize that a conditional is missing, we first ask the user to demonstrate the condition (or let us synthesize a boolean value). She can then demonstrate the new branch as usual. However, when the new branch ends, we must learn where the control rejoins the original partial program. We could ask the user for this information, but it would require her to reason about more than just the current trace (by having to consider the rest of the partial program).

We instead automatically learn where a new branch of a conditional rejoins the original control flow. Once the user marks that the branch has ended, she continues giving the trace as usual. We consider all possible join locations and see to which the user's demonstrations could correspond.

As an example, consider the following partial program.

```
x = 42
y = ?(x + a, x + b)
x = x + y
```

In a new interactive trace, the user might begin by telling us that there is a new conditional and that its body contains a single assignment to `x`. At this point, any of the partial programs below might be valid.

```
if (b)
  x = 137
x = 42
y = -1
x = x+y
```

```
if (b)
  x = 137
else
  x = 42
y = -1
x = x+y
```

```
if (b)
  x = 137
else {
  x = 42
  y = -1
}
x = x+y
```

```
if (b)
  x = 137
else {
  x = 42
  y = -1
  x = x+y
}
```

If the user next demonstrates an assignment to `x` that could correspond to either `x = 42` or `x = x+y`, then only the first and third above partial programs are still possible (if the second partial program were correct, the next action would assign `-1` to `y`, and if the fourth were correct, the trace would be finished). If the next action after that corresponds to `y = -1`, we know the first must be correct, and if it assigns to `x = x+y`, we know the third is correct.

In some cases it might seem that there are no valid join locations because the trace continues into another unseen branch of a missing conditional. For example, consider the following program.

```
if (b1)
  x = 1
else
  x = 2
if (b2)
  y = 1
else
  y = 2
```

If the first trace has `b1 == b2 == true`, then it will assign `1` to both `x` and `y`. If the next trace has `b1 == b2 == false`, when the user demonstrates `x = 2`, we will learn the first conditional. However, the next demonstration will be `y = 2`, which does not correspond to any statement in our current partial program.

In such a case, we can re-execute a previously-seen trace with the knowledge of where the first conditional begins. We can then ask the user to mark the end of the conditional branch as usual, at which point we will have completely discovered the first conditional. Continuing the example, we can re-run the trace where `b1 == b2 == true` with the following partial program.

```
if (b1)
  ?
else
  x = 2
unknown {
  x = 1
  y = 1
}
```

Here, the `unknown` block represents statements that lie either in the then branch or after the conditional. The user will then show that the conditional ends after `x = 1`, at which point we will have the correct code for the first conditional.

Because of this, in the worst case we might need to ask the user for two extra traces for each conditional (the trace where we discovered the other branch and the re-executed trace). However, since this re-executed trace is one the user has already shown us, we will ask no disambiguation questions; the user need only accept our suggestions until she marks the end of the unknown branch.

Discussion. This approach extends to loops. Users can not mark loops when they are initially executed and simply show the executions of their bodies (which might require nothing if the loop is not entered or multiple iterations if it is run multiple times). When a later trace requires a different number of iterations, the user will recognize that there must be a loop and mark it. We have not yet implemented this feature.

An interesting consequence of this lazy approach to inferring control flow is that it might cause us to prune correct possibilities when our partial program does not contain all the

(value)	v		
(identifier)	id		
(range)	r	::=	e `to` e | e `until` e
(lvalue)	l	::=	id | e`[`e`]` | e`.`id
(infix op)	$\oplus$	::=	`+` | `-` | `*` | `/` | `&&` | `||` | `==` | `!=` | `<` | `<=` | `>` | `>=`
(expression)	e	::=	v | l | $e \oplus e$ | `-`e | `!`e | id`(`e`,`e`,`$\cdots$`)`
(statement)	s	::=	s`;`s | e | l `=` e | `if (`e`)` s `else` s | `while (`e`)` s | `for (`id `in` r`)` s | `break`

Figure 5.2: The grammar of the language our implementation supports. We synthesize the expressions and assignments directly and learn the control flow structures with the techniques described in Section 5.3.

conditionals. As an example, consider the following complete program.

```
if (y.left != null)
  x = y.left.value
else
  x = y.value
```

The first trace will only go through one of these branches and might become the following partial program.

```
x = ?(y.left.value, 42)
```

If during pruning we pick an input where `y.left == null`, we will prune away the correct expression `y.left.value`.

To avoid this problem, whenever we discover that we are missing a conditional, we add all of the expressions we have pruned back into the partial program (unless they would have been removed by one of the user's subsequent demonstrations). Once we have added the new conditional, we can run our pruning algorithm again, hopefully removing most of the newly re-added expressions.

5.4 Implementation

We have implemented a prototype of our approach that is available at https://github.com/jgalenson/pbd. We now describe our synthesis algorithm and give details of the techniques described above, including the graphical interface.

Synthesis algorithm. Our implementation operates on a Java-like language that is shown in Figure 5.2 and includes arrays, objects, unary and binary operators, field accesses, function calls, assignments, conditionals, and loops.

As mentioned before, given an action seen during a trace, we synthesize the statements that could have yielded that action and insert them into the partial program. Our algorithm

is a slight modification of the deterministic synthesis algorithm given in Section 3.2.1. We enumerate statements of a certain size (i.e., parse tree height), evaluate them, group them into equivalence classes based on their results, and use representatives of those equivalence classes to generate larger expressions. Once we have reached our fixed maximum size, we keep the expressions that could have yielded the user's action and plug in the equivalent expressions.

As an example, if we determine that `tree` and `x.parent` are equivalent,[2] we will only use `tree` to generate further expressions such as `tree.right` and `foo(tree)`. We will not evaluate `x.parent.right` and `foo(x.parent)` because we know that they are equivalent to expressions we have already generated. If we later find that `tree.right` could have yielded the user's action, we know that `x.parent.right` could have done so as well.

During an interactive trace, if users see that their desired action is not a possibility, they can tell us to search for expressions of a larger size. This leads to an important difference between this algorithm and the one from Section 3.2.1: we must sometimes search for expressions given multiple actions they should yield in different states. Continuing with the previous example, in a second trace the user might ask us to search for larger expressions when `tree` and `x.parent` are no longer equivalent.

The naïve extension of the previous algorithm to handle multiple states would be to do the whole search using equivalence classes created from a single state and then use the other states only to filter out expressions that do not yield the correct result. However, this filtering step must happen after we plug in the equivalent expressions. To see this, consider our running example. If `tree` and `x.parent` are equivalent in the first state but `tree.right` does not yield the correct result in the second state, we would never consider `x.parent.right`, even though in the second state it might yield a different and correct result. Unfortunately, this process of expanding the equivalence classes can exponentially increase the number of expressions we show the user. As an example, if `a`, `b`, `c`, and `d` each have n equivalent expressions, there are n^2 expressions equivalent to `a+b` and n^4 equivalent to `(a+b)+(c+d)`. This naïve algorithm might thus generate exponentially more expressions than are required.

Our algorithm, shown in Algorithm 7, instead generates expressions and equivalence classes separately for each state. For each size, after we generate all expressions of that size, we remove all expressions that crashed in one state, as well as any expressions equivalent to them in that same state, from the equivalences classes of all other memories. Once we have finished generating expressions of the maximum size, we generate the final equivalence classes that represent all states and use them to expand the expressions that yield the desired action in all of the states.

When we infer that a conditional must exist at a certain location in a partial program, we use the information from previous traces at that point to synthesize the condition (or refine the user's demonstration of it). For example, if we discover a conditional on the third trace

[2]Our current implementation is sound in the presence of side effects, as we determine equivalence by comparing the final states as well as the return values.

Algorithm 7 Deterministic synthesis algorithm.

```
for all states σ do
    Initialize classes_σ with null, this, and local variables.
end for
for i ← 2, maxDepth do
    for all states σ do
        newExprs ← genOneLevel(getRepresentatives(classes))
        Evaluate newExprs
        Add newExprs to classes_σ
    end for
    removeCrashers({classes_σ|state σ})
end for
classes ← combineEquivalances({classes_σ|state σ})
result ← filterFailing(getRepresentatives(classes))
result ← expandEquivalences(result)
```

and the user tells us that the condition currently evaluates to `true`, we search for expressions that would have evaluated to `false` when executed in the first two traces and `true` when executed at the current state.

Input generation. We have mentioned that users may supply a generator function that creates inputs to the function to be synthesized. In addition, users may also define filter functions that define whether an input of the right type is a valid input to the function. It has been shown [30] that these two approaches for finding valid inputs are complementary, as in certain cases one is easier to write than the other.

As described in Section 5.1, to find helpful inputs we must find an input whose trace will remove some of the ambiguities in the current partial program. We do this by randomly generating multiple inputs of the correct type and executing them on the partial program as far as possible (i.e., until an ambiguity is reached). If the execution finishes and passes the postcondition, the input will not help. If it finishes and fails the postcondition, we likely need to learn a new conditional or search for expressions of a larger size. As described in Section 5.3, this invalidates any pruning that was already done, so we prioritize this case.

During pruning, we might discover that the program contains no successful paths on a certain input. This also means that we likely need to learn a new conditional or search larger expressions. However, we prefer the previous type of inputs to these, as we know they contain no ambiguities and hence will require us to ask the user fewer disambiguation queries.

If we cannot find an input that suggests our partial program does not contain the correct code, we search for inputs that become stuck at unseens as opposed to holes, since as described in Section 5.2, pruning is more effective when there are fewer unseens. For the same reason, we prefer earlier unseens to later ones. If we find multiple inputs that become stuck

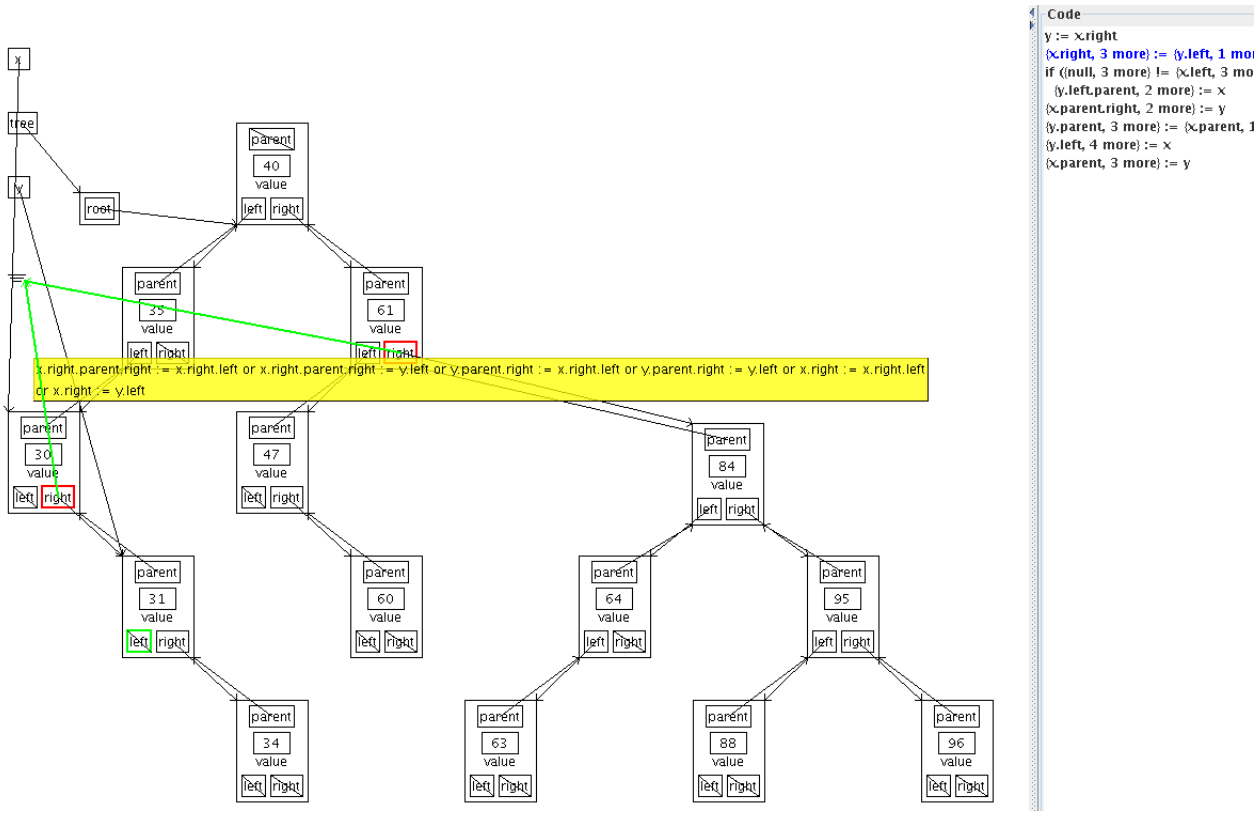


Figure 5.3: An example of our graphical interface during an interactive trace stopped at a hole. Memory is shown on the left and the partial program on the right. The user may click one of the green arrows to select its assignment or drag a green value to one of the red values.

at holes, we prefer the one where the user's answer is likely to resolve the most ambiguities. Our final heuristic is to prefer smaller inputs.

We note that a symbolic or concolic generation technique, e.g., [31], would be more powerful, but we have not yet seen a need to integrate one.

Graphical interface. We present users with a graphical representation of memory with objects containing fields and arrows representing pointers. We additionally show the current code. Users can supply code telling us how to layout objects of certain types.

During a trace, users may manipulate memory bydragging a pointer to a new value (or a special null object), dragging a value on top of another value, clicking on an expression (to evaluate it), or dragging values onto special function call and operator nodes. Users can also add new variables or values, mark the beginning and end of conditionals and loop iterations, and mark the end of the trace.

The GUI enforces type safety: when the user selects and drags a value, it only lets him

drop it on values of a supertype. Such legal targets are colored differently, which also provides visual hints to the user about what to do next.

At each step during an interactive trace, we show the user the current memory and use a different color to show the suggested change (e.g., by showing a differently-colored pointer). When we encounter an ambiguous hole, we show all of the possible actions to the user, with tooltips showing what expressions each represents. For pointer values, the user can simply click on the desired new arrow; for other assignments, he can drag a potential right-hand value to a potential left-hand value. An example is shown in Figure 5.3. If the user is not happy with any of the possible actions, he can tell us to search for expressions of a larger size.

Pruning. Our automated pruning algorithm can be very effective at disambiguating holes. We thus prune whenever the user gives us new code and between all interactive traces. Since pruning is more efficient with fewer unseens, we do it after the user fills in an unseen or gives a new branch of a newly-inferred conditional, which may help automatically resolve ambiguities later in the trace we would otherwise have been forced to ask the user to resolve.

When the user finishes demonstrating the first iteration of a loop, we immediately synthesize code for its body. We then prune the current partial program with unseens inserted after the loop to represent the unknown code. This allows us to reduce the number of queries we ask the user as we walk through the loop (although not as much as we normally can, as we cannot use the postcondition since the program is incomplete).

5.5 Case Studies

We now present some case studies we used to evaluate our methodology. Table 5.1 shows how effectively our approach works for these problems. We do not present any timing information because there were no long pauses. All results were averaged over three runs.

For the first four demonstrations in this section we did not explicitly mark any conditionals. Our algorithm inferred them all easily, requiring no extra traces or queries and learning join points as fast as possible, showing that our inference algorithm is very effective.

Left Rotation. We studied the problem of doing a left rotation for a binary search tree, which was used in the example in Section 2.2. Our postcondition stated that the result is a tree with the same elements as the input.

Since the final code has a conditional with three branches and no loops, any PBD system would require at least three traces to learn the full code. Our system required an average of only 0.7 additional traces to learn the code.

Despite the fact that there was much ambiguity in the graphical demonstrations required for this example, we did not ask the user any disambiguation queries. Our pruning algorithm thus worked very well, and we generated inputs that enabled us to quickly learn the control flow without requiring any disambiguation.

	Minimum	Normal		No pruning	
	Traces	Queries	Traces	Queries	Traces
Rotate	3	0	3.7	9.3	10
Tree Insert	3	0.3	3	3	6
Selection Sort	1	1	2	2	3
List Reversal	1	0	1	5	2.7
RB Insert	2	2	2.6	5	6
DSW	1	2	1	14.3	3.3
Average	1.8	0.9	2.2	6.4	5.2

Table 5.1: Results for our case studies showing the number of queries asked and the total number of unique traces required. The second column shows the minimum number of traces required by any PBD system, the third and fourth columns show the results of our tool, and the last two columns show the same results with pruning disabled.

The usefulness of our pruning can be seen in the last two columns of Table 5.1, which show the same results with pruning disabled. In this case, we ask an average of 9.3 disambiguation queries over ten traces, which requires significantly more work from the user.

Tree Insertion. We analyzed inserting elements into a binary search tree. Our postcondition simply stated that the result is a sorted tree with different elements than the input.

The final code has a conditional with three branches outside the loop, so any PBD system would require at least three traces to learn the code. Our system required exactly three traces and is thus optimal in this respect. We asked 0.3 disambiguation queries on average. This query was inside a loop during the initial trace. As mentioned in Section 5.1, we automatically synthesize the loop body as soon as the first iteration is complete. However, since the full program is not yet complete, we cannot do as much pruning as we would like, so we must ask this extra query.

Without pruning, we ask far more queries in twice as many traces, significantly increasing the burden on the user.

Selection Sort. We analyzed the selection sort algorithm with the postcondition that the output array is sorted. We asked only one disambiguation query, which was needed to determine how to update a loop index variable. This query was asked inside a loop during the initial trace. We needed two traces, one more than optimal, because our initial input did not exercise a branch. Without pruning, we needed an additional trace to ask an additional query.

List Reversal. We analyzed reversing a singly-linked list with the postcondition that the output has the same number of elements as the input. We asked no disambiguation queries in one trace, which is optimal. Without pruning, we required five queries and 2.7 traces to find the same code.

Red-black Tree Insertion. We analyzed inserting elements into a red-black tree. Our postcondition stated that the result is a valid red-black tree.

We asked two disambiguation queries. Even though this code contains many different conditionals inside a loop, our system only required an average of 0.6 traces beyond the minimum. With pruning disabled, we asked on average an additional three queries in 3.4 extra traces.

Deutsch-Schorr-Waite. We analyzed the Deutsch-Schorr-Waite algorithm (DSW) stackless graph marking algorithm [75], which has long been considered difficult for programmers to understand [10]. Our postcondition stated that the graph structure was unchanged and that all the nodes were marked.

We asked two disambiguation queries, both of which were inside a loop in the initial trace, and required only a single trace to learn the program. Without pruning, we asked an average of 14.3 queries in 3.3 traces, requiring significantly more work from the user.

When developing this example, we gave the demonstrations by following some pseudocode of the algorithm. At some points, we misunderstood the missing portions of the code and gave incorrect demonstrations. Our tool automatically found inputs on which our demonstrations were incorrect, which helped us fix our bugs.

Discussion. The above results show that our approach successfully found the code for all of our benchmarks, that our pruning algorithm significantly reduces the amount of work required by the user, and that our conditional inference algorithm is effective. Overall, we ask an average of only 0.9 queries in 2.2 traces (0.2 more than the average minimum required). Without pruning, we ask 6.4 queries in 5.2 traces. We thus reduce the amount of work required of the user by a factor of 2-6 and closely approach the minimum possible.

More than half of the disambiguation queries we asked the user were inside a loop during the initial trace. As mentioned above, while we can still prune the loop and remove some possibilities in such cases, we cannot use the postcondition, as we do not know what will be executed after the loop. Since the alternative is requiring the user to demonstrate the entire loop, we view this as a significant improvement.

In each case our postcondition did not encode the full correctness condition, so we repeated the above experiments with the strongest possible correctness conditions. The results were the same as those reported above, which suggests that our system does not need the strongest possible postcondition. Since disabling pruning is equivalent to pruning with the trivial postcondition that accepts all results, we can see that having some postcondition is very helpful.

In the worst case, we require one initial demonstration of each line but an exponential number of queries to resolve each hole. However, our results show that we do far better in practice. There is only one hole for which we asked more than one disambiguation query, which shows that our queries quickly prune the set of possibilities.

Chapter 6

Related Work

6.1 Approach

Programming by demonstration, also called programming by example, is a popular area of research [39, 24, 55, 50], much of which deals with synthesizing macros [51] and scripts [52]. Unfortunately, these techniques have not been widely adopted, in part because users have difficulty understanding and correcting the learned generalizations [49]. We attempted to avoid these problems by integrating `CodeHint` into the user's workflow and encoding its state directly in the code.

There has been much work on live programming and how it can benefit programmers [94, 18, 86]. This work inspired us to design our methodology to support concrete reasoning.

Program synthesis has had numerous successes at synthesizing code in small well-defined domains such as bitvector logic [47] and data structures [79, 41] as well as somewhat more general classes of programs [81, 48, 82]. As they are backed by decision procedures and SMT solvers, these techniques are very efficient in certain domains that have been fully modeled, while `CodeHint` is not domain-specific and works without any modeling (e.g., it can read from the file system). We thus view these as complementary techniques.

There has been much research on helping programmers explore new APIs by mining existing code to find snippets that are used in practice [59, 44, 74, 88, 37, 36]. Unlike such systems, by evaluating code at runtime, we can differentiate between different values of the same type, downcast precisely, and use more general specifications, as in Section 2.1.

Test generation techniques [91, 95, 77] generate inputs to explore branches within code (which is equivalent to satisfying boolean specifications) but they do not always generate the code to construct those inputs. Seeker [89] synthesizes code fragments with a combination of dynamic and static analysis. Its reliance upon static analysis means it cannot generate certain code fragments that `CodeHint` can but allows it to be more efficient in many cases. We would like to integrate similar techniques into `CodeHint`.

Some existing code search tools [70, 19, 62, 13] allow more general specifications such as natural-language queries or testcases, but they lack the full power of our pdspecs and

our ability to use dynamic information. Our skeletons are similar to the partial expressions of [65] and the holes of [81].

The Smalltalk method finder [14] allows programmers to specify concrete arguments and the desired result and then evaluates all methods of the given receiver with the given arguments to see which return that result. The algorithm is thus very simplistic, but it allows giving multiple demonstrations, similar to our refinement methodology.

In summary, the key benefits of our approach are the precision enabled by our dynamic nature, the generality of our pdspecs, and the fact that we can handle the full Java language without any modeling. The main disadvantages are that we are less efficient and provide fewer correctness guarantees than some existing techniques.

6.2 Algorithm

The way our algorithm iteratively generates expressions of larger depth is similar to the approach used by CHESS [63], as we both search the space in a way that prioritizes elements that are more likely to occur in practice. Our representation of the search space is similar to version space algebra [51, 50] but we can more efficiently eliminate redundant elements.

Our equivalence classes are similar to ideas from Daikon [26], Randoop [64], TRANSIT [92], and model finding [98]. Unlike Randoop, we enumerate all solutions, guided by our probabilistic model toward more likely expressions, up to some bound instead of randomly searching the space. Our techniques are also more powerful than these, and they handle primitives, objects, and arrays. We also apply the algorithms in the new domain of program synthesis, using them to gain the ability to synthesize real Java code. Most importantly, unlike these approaches we are sound in the presence of side effects.

Many existing techniques for undoing side effects in Java focus on transactions [73] or sandboxing [78, 85]. These approaches can be effective but often require modifying the client code or the runtime system. In contrast, our watchpoint-based algorithms are slower but require only a standard JVM. Research on speeding up watchpoints [99] could improve the efficiency of our approach, or we could use a more efficient technique. Some researchers have used standard JVM techniques to reset only static state [23]; our approach in some ways generalizes this work to reset all state.

Some recent work has focused on building probabilistic models for programming languages that are based on natural language. Macho [22] tries to enable program synthesis from natural language specifications by building a probabilistic model that relates code to English. Other work [43] even builds a model of Java code by directly applying statistical models for the English language. By focusing on natural language, it might be easier to incorporate other pieces of information, such as comments in the code.

We would also like to consider other types of probabilistic models. Work that mines API usage protocols [67] might greatly improve our ability to synthesize code involving complicated APIs. Since one of the major benefits of `CodeHint` is that it executes code in a dynamic context, it might make sense to build a model based on traces of executed code

rather than the static source code. Previous efforts have had success using such a model of dynamic traces to aid program synthesis [96, 67].

Our current model was based on analyzing seven projects that together contain over ten million lines of code. While this is not a small corpus, it could be much larger, and the fact that it is drawn from only a small number of projects could introduce biases. There has been much work in mining source code information at a large scale, analyzing hundreds of millions of lines of code in hundreds or thousands of different projects [11, 28], that we would like to leverage, and there are many existing datasets we might even be able to use directly [11, 33, 45]. Our model is more complex than many existing models in that it requires type information. Many of these models are lexical models that simply parse tokens, and even those that parse code into an AST usually do not compute types, although there has been work on inferring types from incomplete snippets [84]. In our case, the type information is useful to differentiate between methods with the same name. Computing a large model with types would be slower and more difficult than using an existing model (we would have to tap into the build system), but might be a useful contribution to the field by itself.

6.3 Programming by Demonstration

Tinker [56, 54] is a PBD system that generates LISP code from concrete traces. It can learn conditionals by noticing discrepancies between multiple examples. However, it cannot handle conditional branches joining as well as we can. Additionally, its demonstrations are not as general as ours; for example, its users must specify any functions they are calling, while in our system users can give the result and we will synthesize the call.

CoScripter [52, 58] is a tool that records user interactions with a webpage so that they can later be replayed as an automated script. Rather than focusing on algorithms, as we do, it is designed to make it easy to share and reuse scripts. We believe it would be useful to combine these ideas.

Some work has focused on using version space algebra [51] to synthesize code from traces [50]. Like ours, this work can generalize traces and learn conditionals and loops. However, it cannot automatically generate new testcases and refine the program being learned, nor can it infer that conditionals are missing in the traces seen so far.

Existing PBD systems deal with ambiguities in different ways. Rather than taking advantage of information about the whole program and its intended behavior as our pruning algorithm does, some approaches simply require enough demonstrations of an individual statement to learn it uniquely [50]. Others develop techniques to rank potential solutions to try to guess the correct expression to present to the user [35, 66]. Some simply make users choose the correct disambiguation [58]. We believe our approach, which uses correctness conditions instead of heuristics, is complementary, and a combination of these techniques seems promising.

PBD has also been used in the HCI field [25], where researchers have studied different ways to resolve ambiguities [60] with an emphasis on usability. We believe a careful study

of this work could improve our own approach and help make it easier to use.

There are many graphical programming languages that allow users to write code visually [71, 93, 69]. Our system uses some of the designs developed in these languages to try to make it easier for users to give us traces.

The Sketch project [81] allowed programmers to write partial programs that it used to synthesize complete code. We have adopted their syntax for our partial programs, but the user experience is vastly different, as its users give partial programs rather than the traces we require. Angelic programming [15] allows users to develop algorithms by examining traces, but it still requires that they write partial programs rather than give traces.

There is a large body of work on program synthesis [82, 47, 81, 79, 41]. Many of these systems generate code from specifications of the output given some hint about the desired code and some formal models; we take advantage of postconditions but only require traces of the algorithm.

Chapter 7

Future Work and Conclusion

We now present our two main proposals for future work: improvements to our probabilistic model (Section 7.1.1) and `CodeHint.js`, the JavaScript version of `CodeHint` (Section 7.1.2). We then briefly mention some other potential avenues in Section 7.1.3.

7.1 Future Work

7.1.1 Probabilistic model of code

We would like to build a database of larger snippets (e.g., ten lines of code instead of a few method calls) that are used in practice. This would allow us to synthesize significantly larger code snippets and bias the search toward code used in practice. We would like to investigate ways of clustering and intersecting the code snippets [19] and plugging them into the current state. We believe that we can combine this model with the smaller-scale models discussed thus far, e.g., by using the smaller models to rank the likelihood of the larger snippets.

A probabilistic model is little help in synthesizing code that involves never-before-seen libraries (perhaps a company's internal library or something the developers wrote themselves). We would like to be able to retain `CodeHint`'s ability to synthesize such code but keep the advantages of using models. One possibility is to update our models based on the user's current project, which would allow us both to analyze any types we have not seen before and let us learn how the current user's code differs from the code in our model.

7.1.2 Improving `CodeHint.js`

Because JavaScript is dynamically typed, the search space of potential code snippets is much larger than for Java, as we cannot use method argument types to reduce the search space. We believe that we can solve this problem by doing type inference. By learning some of the types in the JavaScript code, we will be able to reduce the search space, bringing it closer to the Java version. We can attempt to use an existing type inference algorithm, e.g., [12, 46, 38], or develop our own, perhaps by using simple and targeted heuristics or

adapting solutions to a similar problem in Java [68]. Another possibility is to use techniques based on lazy initialization [7] to infer argument types as we execute code.

Our Java version of `CodeHint` handles side effects by installing breakpoints in the JVM and using them to detect and undo the effects. Since this approach is specific to Java, we will need an alternate approach to handle side effects in Java expressions. We see multiple possible solutions to this problem. We could use a record and replay or dynamic analysis system that supports rolling back values [76]. Newer implementations of JavaScript support techniques similar to the watchpoints used by `CodeHint` [2, 7]. The popular Node.js framework [90] contains a module for executing code in a different context [5] or process [3] or on a cluster [4]. An additional benefit of this approach is that it might make it easy to parallelize our search, making it even more efficient.

The Java version of `CodeHint` is targeted mainly at programmers using new or confusing APIs, and our user study results show that it is useful in such cases. Programmers use JavaScript in different ways than they use Java, however, so `CodeHint.js` might well target different use cases. We would like to investigate how programmers use JavaScript and what sort of tasks they would want solved so that we could design `CodeHint.js` to be of the most help.

7.1.3 Other potential future directions

Doing more with breakpoints. We believe that our breakpoint-based algorithms for detecting and undoing side effects are novel and deserve more attention. Specifically, we would like to continue investigating ways to improve their implementation such as applying instance filters to reduce the number of spurious breaks. We would also like to separate the code from `CodeHint`, which would enable it to communicate with the JVM directly rather than going through the Eclipse APIs. There are also other issues worth considering, such as how to handle writes by multiple threads, and it would be interesting to benchmark our approach and compare it to other techniques.

We also believe that we can use similar approaches for other purposes. For example, we currently have a prototype implementation for `CodeHint` of a form of lazy initialization that uses watchpoints to detect which fields an evaluation touches and uses that to infer which expressions' results can be known without evaluation.

For example, the `JComponent` type (a supertype of most GUI components in the Swing toolkit) contains a `getClientProperty` method that takes in an `Object` and returns a property associated with it. During some of our testing, we ended up calling this method over 100 times with different arguments, but the receiver object contained no properties and so all the calls returned the same value. With this technique, we could detect that a call does not read any field of the current argument and infer that all arguments of the same type would have the same result.

Handling generics. Due to Java's erasure, the type of generic methods and objects is not available at runtime [16]. `CodeHint` thus might generate statements that do not

compile (e.g., `integerList.add("Hi")`), so it currently detects and filters out such cases. In addition, the current implementation might miss expressions that use the erased return value of a method (e.g., we might think a method returns an `Object` when it really is statically known to return a `String`), but it handles this case by downcasting the result to the actual type. We would like to investigate using reflection to get more precise type information to make our search more efficient and precise.

Integrating symbolic techniques. We would like to integrate symbolic techniques into `CodeHint`. Such techniques would complement our approach; our algorithm can handle arbitrary Java method calls while symbolic techniques can quickly explore a very large search space in certain domains.

Methods that can be called a large number of times with different arguments (e.g., a method with four integer arguments when we have seen 100 different integer values) are the worst case for our approach. However, if the method can be modeled by a symbolic technique, e.g., by translating it into SMT, such solvers could likely find multiple solutions very efficiently. We would thus like to consider building a symbolic technique (or using an existing one) that can handle a subset of Java and use it on cases where it would outperform our normal approach.

If properly implemented this approach could greatly increase the space of code snippets that we can search.

Improving usability. We believe that we can improve the usability of `CodeHint` to make it easier for programmers to use. One idea is that instead of simply giving users a (sorted) list of results that satisfy their pdspec, we could group together related expressions. For example, instead of listing the expressions

```
cat.eat(), cat.meow(), foo.bar(a), foo.bar(b)
```

we could show the user something like

```
cat.??{eat,meow}(), foo.bar(??{a,b})
```

that represents all of the similar possibilities. This would help users manage a large number of results.

This technique could also help make our search more interactive. Instead of doing a search and then presenting users with its results, we could show them, in a fashion similar to the snippet with holes above, the possible expressions we might search and let them choose which we should explore further. In the above example, the user might mark that we should explore expressions that involve the `cat` methods but not those that call `foo.bar`. This technique might allow us to work with the user to synthesize large code fragments.

Automatically generating benchmarks. We currently evaluate `CodeHint` on a handful of manually-created benchmarks. However, using only this small set makes it difficult to evaluate changes to the algorithm, as we run the risk of overfitting solutions.

To alleviate this problem, we would like to develop a methodology for automatically generating benchmarks. As opposed to existing approaches such as bug finding (which can look for bugs in real-world programs) and probabilistic models (which can test how many methods would be correctly identified by autocomplete), the generality of our pdspecs makes this a difficult problem.

One possible solution is to automatically run tests for certain codebases, extract the types and values certain expressions return, and use those as the pdspecs and desired code. We could handle refinement by generating multiple such input-output pairs. For more complicated pdspecs, we could look for `asserts` or unit tests.

Programming by Demonstration. Our current approach requires that the user demonstrate each statement in a trace in the correct order. We would instead like to allow users to show a desired final or intermediate state. This would enable users to give more modular traces; they could initially demonstrate some parts of a trace in detail while giving only the end result of executing a certain chunk of code.

To be able to demonstrate a trace to our tool, users must understand how their algorithm processes concrete inputs. To remove this limitation, we would like to support users who have only incomplete knowledge of how their desired algorithm proceeds, e.g., by suggesting traces to the user.

We believe that graphical abstractions could be very useful for demonstrating algorithms in different paradigms (e.g., functional) and applications (e.g., cryptography).

7.2 Conclusion

We have presented a novel methodology that helps programmers write difficult statements. This methodology is dynamic (evaluating code and runtime and letting users inspect the results), easy-to-use (accepting a wide range of specifications), and interactive (helping users gain information about the missing code). Our algorithms are efficient and let us synthesize code that uses real-world Java features such as native calls and reflection. We have run two user studies that shows that our tool `CodeHint` significantly improves programmer productivity. We have also presented a novel programming methodology that synthesizes code from user's demonstrations of concrete traces of their algorithms. We have developed automatic techniques to reduce the burden on the user and have shown that they are effective in practice.

Bibliography

[1] Darts, Dice, and Coins: Sampling from a Discrete Distribution. http://www.keithschwarz.com/darts-dice-coins/. Accessed: 6/19/2014.

[2] Harmony observe. http://wiki.ecmascript.org/doku.php?id=harmony:observe. Accessed: 9/23/2013.

[3] Node.JS child_process. http://nodejs.org/api/child_process.html. Accessed: 9/23/2013.

[4] Node.JS cluster. http://nodejs.org/api/cluster.html. Accessed: 9/24/2013.

[5] Node.JS vm. http://nodejs.org/api/vm.html. Accessed: 9/23/2013.

[6] Object. http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html. Accessed: 6/20/2014.

[7] Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Accessed: 9/23/2013.

[8] TIOBE Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 9/20/2013.

[9] Unreal Engine 3. http://www.unrealengine.com/html5/. Accessed: 07/03/2013.

[10] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer Berlin Heidelberg, 2003.

[11] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[12] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented*

Programming, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer Berlin Heidelberg, 2005.

[13] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 681–682, New York, NY, USA, 2006. ACM.

[14] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, Marcus Denker, et al. *Pharo by example*. 2009.

[15] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 339–352, New York, NY, USA, 2010. ACM.

[16] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM.

[17] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[18] M. M. Burnett, J. W. Atwood Jr, and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of the IEEE Symposium on Visual Languages*, VL '98, pages 126–, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

[21] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997.

[22] Anthony Cozzie and Samuel T King. Macho: Writing programs with natural language and examples. Technical report, University of Illinois at Urbana-Champaign, 2012.

[23] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.

[24] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, 1993.

[25] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. A cappella: Programming by demonstration of context-aware applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 33–40, New York, NY, USA, 2004. ACM.

[26] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[27] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.

[28] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[29] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[30] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in udita. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM.

[31] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[32] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[33] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[34] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8 –21, oct. 1978.

[35] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[36] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 27–38, New York, NY, USA, 2013. ACM.

[37] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 418–423. Springer-Verlag, Berlin, Heidelberg, 2011.

[38] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[39] Daniel Conrad Halbert. *Programming by example*. PhD thesis, 1984.

[40] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:366–381, 2000.

[41] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 38–49, New York, NY, USA, 2011. ACM.

[42] Michi Henning. Api design matters. *Queue*, 5(4):24–36, May 2007.

[43] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[44] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[45] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. An unabridged source code dataset for research in software reuse. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 339–342, Piscataway, NJ, USA, 2013. IEEE Press.

[46] SimonHolm Jensen, Anders MÃÿller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer Berlin Heidelberg, 2009.

[47] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[48] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM.

[49] Tessa Lau. Why pbd systems fail: Lessons learned for usable ai. In *Computer Human Interaction*, 2008.

[50] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2Nd International Conference on Knowledge Capture*, K-CAP '03, pages 36–43, New York, NY, USA, 2003. ACM.

[51] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 527–534, San Francisco, CA, USA, 2000.

[52] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[53] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[54] Henry Lieberman. Watch what i do. chapter Tinker: a programming by demonstration system for beginning programmers, pages 49–64. MIT Press, Cambridge, MA, USA, 1993.

[55] Henry Lieberman, editor. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., 2001.

[56] Henry Lieberman and Carl Hewitt. A session with tinker: Interleaving program testing with program design. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 90–99, New York, NY, USA, 1980. ACM.

[57] J. L. Lions. Ariane 5 Flight 501 Failure – Report by the Inquiry Board. Technical report, 1996.

[58] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. Koala: Capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 943–946, New York, NY, USA, 2007. ACM.

[59] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[60] Jennifer Mankoff, Scott E. Hudson, and Gregory D. Abowd. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, UIST '00, pages 11–20, New York, NY, USA, 2000. ACM.

[61] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.

[62] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[63] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[64] Carlos Pacheco. *Directed random testing*. PhD thesis, 2009.

[65] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 275–286, New York, NY, USA, 2012. ACM.

[66] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 408–418, New York, NY, USA, 2014. ACM.

[67] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[68] Michael Pradel, Severin Heiniger, and Thomas R. Gross. Static detection of brittle parameter typing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 265–275, New York, NY, USA, 2012. ACM.

[69] Miller Puckette. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference*, pages 37–41, 1996.

[70] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[71] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009.

[72] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[73] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 439–448, Washington, DC, USA, 2002. IEEE Computer Society.

[74] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 413–430, New York, NY, USA, 2006. ACM.

[75] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, August 1967.

[76] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[77] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[78] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[79] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 289–299, New York, NY, USA, 2011. ACM.

[80] David Canfield Smith, Allen Cypher, and Larry Tesler. Programming by example: Novice programming comes of age. *Commun. ACM*, 43(3):75–81, March 2000.

[81] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.

[82] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

[83] Arthur G Stephenson, Daniel R Mulville, Frank H Bauer, Greg A Dukeman, Peter Norvig, LS LaPiana, PJ Rutledge, D Folta, and R Sackheim. Mars climate orbiter mishap investigation board phase i report, 44 pp. *NASA, Washington, DC*, 1999.

[84] Siddharth Subramanian and Reid Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 85–88, Piscataway, NJ, USA, 2013. IEEE Press.

[85] Mengtao Sun and Gang Tan. Jvm-portable sandboxing of javaâĂŹs native libraries. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security âĂŞ*

ESORICS 2012, volume 7459 of *Lecture Notes in Computer Science*, pages 842–858. Springer Berlin Heidelberg, 2012.

[86] Steven L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, June 1990.

[87] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[88] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[89] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 189–206, New York, NY, USA, 2011. ACM.

[90] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *Internet Computing, IEEE*, 14(6):80–83, 2010.

[91] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[92] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 287–296, New York, NY, USA, 2013. ACM.

[93] Lisa K. Wells and Jeffrey Travis. *LabVIEW for everyone: graphical programming made even easier*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[94] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '97, pages 258–265, New York, NY, USA, 1997. ACM.

[95] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381. Springer-Verlag, Berlin, Heidelberg, 2005.

[96] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 65–82, New York, NY, USA, 2011. ACM.

[97] Alon Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 301–312, New York, NY, USA, 2011. ACM.

[98] Jian Zhang and Hantao Zhang. Sem: A system for enumerating models. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 298–303, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[99] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 147–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[100] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

Appendix A

Benchmark programs

We list below some of the code for our benchmarks. The first three were used in the first user study and our empirical evaluation while the last was used in our second user study. The other benchmark used in the second user study is split over multiple different files so we do not list it here. Its tasks involved serializing objects to/from strings, handling graphical clicks, and implementing bold, italics, and underline. The code for this missing task as well as the full code for the other examples can be viewed at https://github.com/jgalenson/codehint/tree/master/examples.

The code snippets below show the benchmarks as shown to the subjects with the solution code commented out. Tasks are denoted by `println`s or comments.

Listing A.1: The Parse program.

```
1  import java.util.HashMap;
2  import java.util.List;
3  import java.util.Map;
4  import java.util.Arrays;
5
6  import codehint.CodeHint;
7
8  @SuppressWarnings("unused")
9  public class Parse {
10
11     /*
12      * Builds a Map that, for each line in the string, has whatever is
13      * before the separator map to whatever is after it.
14      * For example, given the email below, it will map "Subject" to
15      * "[cs-grads-food] Pizza" and "Date" to "Mon, 27 Feb 2012 15:28:03 -0800".
16      */
17     private static Map<String, String> parse(String email, String separator) {
18         String[] lines = email.split("\n");
19         Map<String, String> parts = new HashMap<String, String>(lines.length);
20         for (String line : lines) {
21             int index = 0;
22             //index = line.indexOf(separator);
```

```
23          System.out.println("Task: Find the first index where the separator string
                occurs in the line.");
24
25          String header = null;
26          //header = line.substring(0,index);
27          System.out.println("Task: Find everything that comes before the separator
                in the line.");
28
29          String body = null;
30          //body = line.substring(index + separator.length());
31          System.out.println("Task: Find everything that comes after the separator
                in the line.");
32
33          parts.put(header, body);
34      }
35      System.out.println(parts);
36      return parts;
37  }
38
39  /*
40   * Find the argument that comes offset arguments after the target string, if it
         is in the array.
41   * For example, getArgument(new String[] {"a", "b", c", "d"}, "a", 2) returns "c
         ".
42   */
43  private static String getArgument(String[] argsArr, String target, int offset) {
44      List<String> argsList = null;
45      //argsList = Arrays.asList(argsArr);
46      System.out.println("Task: Convert the array of arguments into a list.");
47
48      int index = 0;
49      //index = argsList.indexOf(target);
50      System.out.println("Task: Get the index of the target string in the array/list
            .");
51
52      if (index == -1)
53          return null;
54      int newIndex = index + offset;
55      if (newIndex >= argsArr.length)
56          return null;
57      return argsList.get(newIndex);
58  }
59
60  private static final String TEST_EMAIL = "Message-ID: <4F4C1183.3030805@cs.
        berkeley.edu>\nDate: Mon, 27 Feb 2012 15:28:03 -0800\nFrom: Redacted\nUser-
        Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:10.0.2) Gecko/20120216
        Thunderbird/10.0.2\nMIME-Version: 1.0\nTo: cs-grads-food@eecs.berkeley.edu\
        nSubject: [cs-grads-food] Pizza";
61
62  public static void main(String[] args) {
```

```
63          parse(TEST_EMAIL, ": ");
64          getArgument(new String[] { "--foo", "--bar", "--baz", "42" }, "--baz", 1);
65          getArgument(new String[] { "--foo", "--bar", "--baz", "42" }, "--foo", 0);
66          getArgument(new String[] { "--foo", "--bar", "--baz", "42" }, "--bar", 0);
67      }
68
69  }
```

Listing A.2: The Swing program.

```
1  import java.awt.Dimension;
2  import java.awt.Window;
3  import java.awt.event.MouseAdapter;
4  import java.awt.event.MouseEvent;
5
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JTree;
9  import javax.swing.SwingUtilities;
10 import javax.swing.tree.TreePath;
11
12 import codehint.CodeHint;
13
14 @SuppressWarnings("unused")
15 public class SwingTest {
16
17     private static void createAndShowGUI(JFrame frame) {
18         Dimension dim = null;
19         //dim = frame.getPreferredSize();
20         System.out.println("Task: Find a Dimension that somehow represents the size of
               the frame.");
21
22         int w = (int)dim.getWidth();
23         int h = (int)dim.getHeight() / 2;
24         Dimension newDim = null;
25         //newDim = new java.awt.Dimension(w,h);
26         System.out.println("Task: Make a Dimension with the given width and height.");
27
28         JLabel label = new JLabel("Hello, world!");
29         MySwingHelper.resizeLabel(label, newDim);
30         frame.add(label);
31
32         Window window = null;
33         //window = frame;
34         System.out.println("Task: Find a Window that contains the label.");
35         window.setAlwaysOnTop(true);
36     }
37
38     /*
39      * A JTree displays data in a hierarchical form, where child nodes can be hidden
            or expanded as necessary.
```

```
40      * When run, this code will make a small little JTree that will pop up on the
            screen.
41      */
42     private static void createJTree(JFrame frame) {
43         final JTree jtree = MySwingHelper.createTree();
44 
45         /*
46          * This adds a mouse handler that triggers whenever the use clicks on the tree
                .
47          */
48         jtree.addMouseListener(new MouseAdapter() {
49             public void mousePressed(MouseEvent e) {
50                 JTree tree = jtree;
51                 int mouseX = e.getX();
52                 int mouseY = e.getY();
53                 int row = 0;
54                 //row = tree.getRowForLocation(mouseX,mouseY);
55                 System.out.println("Task: Figure out which row (0-based, containing all
                        elements in the tree, including the top-level one) the user
                       clicked, or -1 if they didn't click a valid element.");
56 
57                 TreePath path = null;
58                 //path = tree.getPathForRow(row);
59                 System.out.println("Task: Find the path the user clicked, or null if
                       they didn't click a valid element.");
60 
61                 System.out.println(row);
62                 System.out.println(path);
63             }
64         });
65 
66         frame.add(jtree);
67     }
68 
69     public static void main(String[] args) {
70         SwingUtilities.invokeLater(new Runnable() {
71             public void run() {
72                 JFrame frame = MySwingHelper.makeFrame("Swing test");
73                 createAndShowGUI(frame);
74                 createJTree(frame);
75                 MySwingHelper.showFrame(frame);
76             }
77         });
78     }
79 
80 }
```

Listing A.3: The RandomWriter program.

```
1 import java.util.ArrayList;
2 import java.util.Collection;
```

```
3  import java.util.HashMap;
4  import java.util.List;
5  import java.util.Map;
6  import java.util.Random;
7
8  import codehint.CodeHint;
9
10 public class RandomWriter {
11
12     /*
13      * Make a model that computes, for each string of size order in text (called a
            seed),
14      * all the characters that follow that seed.
15      * As an example, given "abacabc", with order 1, "a" is followed by "b", "b", and
            "c";
16      * with size 2, "ab" is followed by "a" and "c", and so forth.
17      * Note that this is case-sensitive.
18      */
19     private static Map<String, List<Character>> makeMarkovModel(String text, int
           order) {
20         Map<String, List<Character>> model = new HashMap<String, List<Character>>();
21         /* Loop over all the substrings of length order and update the information
               about the characters that follow them. */
22         for (int i = 0; i < text.length() - order; i++) {
23             int j = i + order;
24
25             String seed = null;
26             //seed = text.substring(i,j);
27             System.out.println("Task: Find the seed string.");
28
29             char follower = 0;
30             //follower = text.charAt(j);
31             System.out.println("Task: Find the character that follows the seed.");
32
33             /* Add this following character to the list of characters we've seen that
                   follow this seed. */
34             if (!model.containsKey(seed))
35                 model.put(seed, new ArrayList<Character>());
36             model.get(seed).add(follower);
37         }
38         return model;
39     }
40
41     /*
42      * Find the seed that occurs most often in the model.
43      * Ties are broken arbitrarily.
44      */
45     private static String getMostCommonSeed(Map<String, List<Character>> model) {
46         String best = null;
47
```

```
48          Collection<String> allKeys = null;
49          //allKeys = model.keySet();
50          System.out.println("Task: Find a Collection of all the keys in the model.");
51
52          /* Given the set of keys, find one that occurs the most often. */
53          for (String seed: allKeys)
54              if (best == null || model.get(seed).size() > model.get(best).size())
55                  best = seed;
56          return best;
57      }
58
59      /*
60       * Randomly generate a string similar to the original input text by starting
61       * with the most commonly-occurring seed and updating that with a character
             chosen
62       * based on the probabilities of what followed that seed in the original text.
63       */
64      private static String doRandomWrite(Map<String, List<Character>> model, int
             length) {
65          String seed = null;
66          //seed = getMostCommonSeed(model);
67          System.out.println("Task: Get one of the seeds that occurs the most in the
                 input string / model.");
68
69          String text = seed;
70          Random random = new Random();
71          /* Generate the next character of the text and update the seed to include it.
                 */
72          while (text.length() < length && model.containsKey(seed)) {
73              List<Character> followers = model.get(seed);
74
75              int rand = 0;
76              //rand = random.nextInt(followers.size());
77              System.out.println("Task: Get a random index inside the list of followers.
                     ");
78
79              char next = followers.get(rand);
80              text += next;
81              seed = seed.substring(1) + next;
82          }
83          return text;
84      }
85
86      public static void main(String[] args) {
87          Map<String, List<Character>> model = makeMarkovModel("This is a test of the
                 emergency broadcast system.", 2);
88          String commonSeed = getMostCommonSeed(model);
89          System.out.println(doRandomWrite(model, 100));
90      }
91
```

```
92 }
```

Listing A.4: The Eclipse program.

```
1  package memanalyzer.handlers;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import memanalyzer.ReportDialog;
7  import memanalyzer.SizeUtils;
8  import memanalyzer.StackInfo;
9  import memanalyzer.TypeInfo;
10
11 import org.eclipse.core.commands.AbstractHandler;
12 import org.eclipse.core.commands.ExecutionEvent;
13 import org.eclipse.core.commands.ExecutionException;
14 import org.eclipse.debug.core.DebugException;
15 import org.eclipse.debug.core.model.IStackFrame;
16 import org.eclipse.debug.core.model.IThread;
17 import org.eclipse.debug.core.model.IVariable;
18 import org.eclipse.debug.ui.DebugUITools;
19 import org.eclipse.swt.widgets.Shell;
20 import org.eclipse.ui.handlers.HandlerUtil;
21 import org.eclipse.jdt.debug.core.IJavaStackFrame;
22
23 import codehint.CodeHint;
24
25 import com.sun.jdi.ArrayReference;
26 import com.sun.jdi.ArrayType;
27 import com.sun.jdi.ClassType;
28 import com.sun.jdi.InterfaceType;
29 import com.sun.jdi.ObjectReference;
30 import com.sun.jdi.ReferenceType;
31 import com.sun.jdi.VirtualMachine;
32
33 @SuppressWarnings("unused")
34 public class MemoryAnalyzerHandler extends AbstractHandler {
35
36     /*
37      * This method is called when the user clicks the
38      * "Analyze Memory" item. It is a standard part of
39      * Eclipse plugins; in fact, the plugin creation wizard
40      * created an empty version of it for us.
41      */
42     @Override
43     public Object execute(ExecutionEvent event) throws ExecutionException {
44         try {
45             /*
46              * TASK 1
47              * ------
```

```
48              * Find the IThread representing the current
49              * thread and use it to create a StackInfo
50              * object that represents the size of the
51              * stack.
52              *
53              * Note that you should only do this for the
54              * current thread (i.e., the one that is
55              * stopped at the breakpoint), not for all
56              * threads.
57              */
58             IThread curThread = null;
59             StackInfo stackInfo = null;
60             /*curThread = ((IJavaStackFrame)DebugUITools.getDebugContext()).getThread
                   ();
61             IStackFrame[] stacks = curThread.getStackFrames();
62             int totalStackSize = 0;
63             for (IStackFrame stack: stacks)
64                 totalStackSize += SizeUtils.getStackFrameSize(stack);
65             stackInfo = new StackInfo(stacks.length, totalStackSize);*/
66
67
68             /*
69              * TASK 2
70              * ------
71              * Find the VirtualMachine representing the
72              * running VM and use it to get all the loaded
73              * types. For each loaded type with at least
74              * one instance, create a TypeInfo object that
75              * represents the total size of objects of
76              * exactly that type in the heap. Also compute
77              * the total size of all objects in the heap.
78              *
79              * Note that there are multiple ways to get the
80              * number of instances of a given type that can
81              * be inconsistent. Don't worry about that;
82              * just choose one.
83              */
84             VirtualMachine vm = null;
85             List<TypeInfo> typeInfos = new ArrayList<TypeInfo>();
86             long totalHeapSize = 0;
87             /*vm = ((org.eclipse.jdt.internal.debug.core.model.JDIDebugTarget)
                   curThread.getDebugTarget()).getVM();
88             List<ReferenceType> types = vm.allClasses();
89             long[] instanceCounts = vm.instanceCounts(types);
90             for (int i = 0; i < types.size(); i++) {
91                 ReferenceType type = types.get(i);
92                 long instances = instanceCounts[i];
93                 if (instances == 0)
94                     continue;
95                 long typeSize = SizeUtils.getTotalSizeOfType(type, instances);
```

```
96                totalHeapSize += typeSize;
97                typeInfos.add(new TypeInfo(type.name(), instances, typeSize));
98            }*/
99
100
101           /*
102            * TASK 3
103            * ------
104            * Create and open a ReportDialog to show the
105            * information to the user.
106            */
107           ReportDialog myDialog = null;
108           /*Shell shell = null;
109           shell = HandlerUtil.getActiveShell(event);
110           ReportDialog myDialog = new ReportDialog(shell, typeInfos, totalHeapSize,
                  stackInfo);*/
111
112           myDialog.open();
113       } catch (DebugException e) {
114           throw new RuntimeException(e);
115       }
116       return null;
117   }
118
119 }
```