

©Copyright 2015

Daniel Perelman

Program Synthesis
Without Full Specifications
for Novel Applications

Daniel Perelman

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Daniel Grossman, Chair

Sumit Gulwani, Chair

Emina Torlak

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Program Synthesis
Without Full Specifications
for Novel Applications

Daniel Perelman

Co-Chairs of the Supervisory Committee:
Associate Professor Daniel Grossman
Computer Science & Engineering
Affiliate Professor Sumit Gulwani
Computer Science & Engineering

Program synthesis is a family of techniques that generate programs from a description of what the program should do but not how it should do it. By designing a program synthesis algorithm together with the user interaction model we show that by accepting small increases in user effort, it is easier to write the synthesizer and the need for specialization of the synthesizer to a given domain is reduced without losing performance. In this work, we target three tasks to show the breadth of our methodology: code completion, end-user programming-by-example for data transformations, and feedback for introductory programming assignments. For each of these tasks, we develop an interaction model and program synthesis algorithm together to best support the user. In the first, we use partial expressions to allow programmers to express exactly what they don't know and want the completion system to fill in. In the second, we use the sequence of examples to inform building up larger programs iteratively. In the last, we use attempts from other students on the same assignment to mine corrections.

CONTENTS

Contents	i
List of Figures	iv
List of Tables	vi
List of Algorithms	vii
Chapter 1: Introduction	1
1.1 Contributions	2
1.2 Partial Expression Completer	2
1.3 Test-Driven Synthesis	4
1.4 Code Hunt hint system	5
1.5 Outline	6
Chapter 2: Background	7
2.1 Program synthesis	7
2.2 API discovery	16
2.3 Programming by example	21
2.4 Automated grading	23
2.5 Lessons learned	27

Chapter 3: Partial Expression Completer	29
3.1 Illustrative examples	30
3.2 Partial expression language	36
3.3 Ranking	40
3.4 Algorithm	46
3.5 Evaluation	50
3.6 Conclusion	69
Chapter 4: Test-Driven Synthesis	70
4.1 Motivating examples	71
4.2 Language	75
4.3 Test-Driven Synthesis	80
4.4 DSL-Based Synthesis	86
4.5 Evaluation	95
4.6 Conclusion	108
Chapter 5: Code Hunt hint system	109
5.1 The Code Hunt game	110
5.2 The Feedback System	113
5.3 Algorithm	115
5.4 Evaluation	122
5.5 Conclusion	128
Chapter 6: Conclusions and future work	129

Bibliography	131
------------------------	-----

LIST OF FIGURES

3.1	Workflow of Partial Expression Completer	30
3.2	Results of example method call query	31
3.3	Results of example method argument query	32
3.4	Results of example field lookup query	33
3.5	Partial expression language	37
3.6	Semantics of partial expressions	39
3.7	Ranking function for partial expression completer	41
3.8	The method index	48
3.9	Rate of predicting method calls	53
3.10	# of arguments needed to predict method calls of different # of arguments	53
3.11	Comparison to code completion	54
3.12	Comparison to code completion filtered by return type	55
3.13	Running time of partial expression completer	57
3.14	Rate of predicting arguments	58
3.15	Kinds of expressions in argument positions	59
3.16	Rate of predicting field lookups in assignments	60
3.17	Rate of predicting field lookups in comparisons	61
3.18	Performance of learned ranking functions on the test set	66

3.19	By query comparison of the baseline vs. learned ranking function.	67
3.20	The learned decision tree ranking function	68
4.1	LaSy program for greedy word wrap	72
4.2	LaSy program for converting bibliography entries	74
4.3	LaSy program for converting set of XML lists to a table	76
4.4	LaSy program for adding class attributes	76
4.5	The LaSy language.	76
4.6	The extended FlashFill DSL	78
4.7	Example loop strategies	91
4.8	Comparison between TDS and humans playing Pex4Fun by time	102
4.9	Comparison between TDS and humans playing Pex4Fun by # of tests used .	103
4.10	Time TDS took to find a solution given reordered examples	104
4.11	Proportion of reorderings TDS failed on	105
4.12	Effectiveness of algorithm with parts disabled	106
4.13	Execution times of all DBS runs.	107
5.1	The Code Hunt test results screen.	110
5.2	The Code Hunt “Captured!” screen.	111
5.3	CDF of time between first and last play by A/B test condition	126
5.4	CDF of # of levels won by A/B test condition	126
5.5	Number of attempts which got each hint kind by edit distance	127
5.6	Proportion of attempts which got each hint kind by edit distance	128

LIST OF TABLES

3.1	Summary of quality of best results for each call	52
3.2	Ranking function term sensitivity.	63
3.3	Features for learned ranking function	65
4.1	End-user data transformation benchmark results.	98
5.1	Different kinds of feedback in Code Hunt	112

LIST OF ALGORITHMS

3.1	Algorithm for partial expression completer without optimizations	46
4.1	Test-Driven Synthesis algorithm	81
4.2	DSL-based synthesis algorithm (Test-Driven Synthesis inner loop)	87

ACKNOWLEDGMENTS

Thanks to my advisors Dan Grossman and Sumit Gulwani for their help and guidance throughout the past five years. Thanks also to all of my other co-authors—Tom Ball, Peter Provost, and Judith Bishop—as well as to the rest of the Code Hunt team, Peli de Halleux and Nikolai Tillman. Thanks also to the other members of my committee for their feedback: Emina Torlak and Gary Hsieh.

Chapter 1

INTRODUCTION

A program synthesizer [14] is a system that generates a program given some sort of description of goals or specification of the program. This description may take many forms including complete logical specifications [27], programs [13, 49], incomplete programs [52], and input/output examples [17, 39, 20] or demonstrations of the desired behavior [9, 33].

The reader is likely familiar with another class of systems that generate programs: compilers. We could simply define compilers as a special case of program synthesizers and move on, but that would be missing some important nuance. Compilers transform a program from one language to another, often applying optimizations along the way, but their output is expected to implement more or less the same algorithm as their input. The input program is used as a description of how the output program should work. On the other hand, a program synthesizer treats its input as a description of what the output program should do; in the case of the input being a program, the output may use a completely different algorithm.

In this work, we are particularly interested in looking at what form the input to a program synthesis system takes and how that informs its design. Given a task which we wish to support using program synthesis, we want to design a workflow simultaneously considering the ease of use of the user interface and the usefulness of the information communicated through that interface for performing the program synthesis. To give a concrete example, one of the systems which will be discussed later performs programming by example, but in addition to a set of examples, it takes as input their order, taking advantage of the fact that it is natural for users to provide examples in order of complexity. By using this additional information, the synthesis algorithm was made vastly more efficient while requiring very little

additional effort from the user.

At first, this design methodology may seem like too much effort. After all, if for every task we discover some new form of input that we need to build up a new algorithm to handle, then perhaps it's not worth it. Instead, we find that once we choose the right input, relatively simple synthesis algorithms suffice to achieve good performance; all of the systems we develop are based on enumerative synthesis with optimizations. In order to avoid distracting from our focus on inputs, we aim for simplicity in our algorithms where possible; although we do not explore the possibility, we hope combining our input selection ideas with more advanced algorithms would even further improve performance. Additionally, due to not overspecializing our algorithms to specific domains, our synthesizers can be used in multiple domains.

There is a multi-way trade-off between user effort using the synthesizer, programmer effort writing the synthesizer, specialization of the synthesizer to the domain, and performance (both recall and speed) of a synthesizer. But this trade-off is not balanced. *Small increases in user effort can lead to great savings in both programmer effort and need for specialization of a program synthesizer to a given domain without losing performance.* Note that while we conjecture that our chosen inputs require little effort, we did not confirm that with user studies.

1.1 Contributions

The primary contribution of this work is the insight that the input to a program synthesizer is worth careful attention. This is backed up by three projects in the areas of API discovery, end-user programming by example, and automated grading that take advantage of this insight and demonstrate that it is widely applicable.

1.2 Partial Expression Completer

Modern software development is supported by huge libraries providing hundreds of thousands of methods that programmers can use as they write their own code. The trade-off for having

so much useful functionality available is that discovering a desired API may be difficult and time-consuming, even if it exists.

The normal techniques for dealing with this complexity are to use web searches or to browse through the available methods using the code completion feature available in most modern Integrated Development Environments (IDEs). The former has the problems of both requiring an interruption in the code writing process to switch to a web browser and needing the programmer to be able to phrase the question in a way someone online has already answered; if the library is not popular enough or, worse, not publicly available, then that second condition will definitely fail. On the other hand, code completion has easy integration into the coding process, but is limited to finding methods when the programmer more or less knows where to find them.

More recent research has produced various augmented code completion schemes that support different ways of searching through libraries while staying within the IDE setting. Inputs for these systems include keywords [34], input and output types for an expression [35, 54] and simply using the available local variables to form any valid expression for the return type at the cursor [5, 19, 18].

We instead looked at what format would maximize the knowledge the programmer could provide to the tool while staying within the mode of writing code. To that end, we developed a system based on *partial expressions* which are a superset of the language being programmed in that they allow for parts of the expression to be left blank for the system to suggest completions for. This is expressive enough to cover the queries mentioned above like generating an expression of a given type using the available local variables or a value of a specific input type as well as allowing for more expressive queries like finding a method that takes two specific values as arguments.

We assert that writing API discovery queries as partial expressions requires minimal user effort because they are already thinking in terms of writing an expression in the target programming language. Additionally, it lets them express everything they know (statically) about the desired expression, so it makes the synthesizer's job easier by narrowing the search

space. By not using keywords or natural language, our system does not rely on human written documentation or web search results, so it works with any library.

Chapter 3 discusses the Partial Expression Completer project in more detail.

1.3 Test-Driven Synthesis

Programming-by-example (PBE [9, 33]) empowers end-users without programming experience to automate tasks like normalizing table layouts or reformatting strings merely by providing input/output examples. Present PBE technologies are designed for a specific set of data types. This can be done by using an SMT solver or similar technology [52, 60] and being limited to efficiently handling only those data types whose operations map well to the SMT solver’s theories. Another way is to create a domain-specific language (DSL) for the task in concert with a program synthesis algorithm capable of producing programs in that DSL [17, 20, 29].

With that in mind, we sought to develop a new algorithm that would be able to work across multiple domains without relying on solver support for that domain or a carefully constructed DSL. To do so, we needed to consider what new information from the user we would be using to drive this new algorithm. Inspired by the claims of some proponents of test-driven development (TDD) [22, 36] that the test case order is the hard part of TDD-style programming while the actual code writing is more or less mechanical, we chose to use the order of the examples as an input. As humans naturally describe complex task with examples of increasing difficulty, it seemed likely this order could be valuable to an algorithm.

“The Transformation Priority Premise” [36] proposes that TDD should be considered as a process where the test case sequence corresponds to a sequence of small mutations to the program being developed. One test case might lead a programmer to refining an expression, another might require new control flow. That blog post demonstrates that a poor order will require larger changes at some steps while a good order will only involve small program changes, so the order matters. Following this model, we developed the Test-Driven Synthesis (TDS) program synthesizer which takes as input a DSL and a sequence of

examples and iteratively mutates the synthesized program to satisfy the next example in the sequence until finally outputting a program that satisfies all of the examples.

In addition to being easy for a human to provide, the order of examples is enough information for our iterative enumerative synthesis algorithm to perform well in multiple domains without needing to be specialized to them, thereby both increasing generality and avoiding the programmer effort of implementing specialized synthesizers.

Chapter 4 describes Test-Driven Synthesis in more detail.

1.4 Code Hunt hint system

Even for the small programs given in an introductory computer science course, grading is difficult, or at least tedious and error-prone. The problem is only magnified by the recent popularity of massive open online courses (MOOCs) and websites for learning programming [21]. Due to the scale of the courses, human grading is infeasible while traditional automated techniques (running the submitted program on a fixed test suite) provide very limited feedback. Particularly for students new to programming, a counterexample from a failing test suite may be insufficient to guide a student to understanding the error. A human grader may be able to identify the student's mistake and explain it to the student; we aim to use program synthesis to correct student mistakes in order to provide similar feedback automatically.

We take the Autograder [51] model of the problem: like Autograder, we reduce the feedback problem to a program synthesis problem by looking for a solution to the assignment as close as possible to the student's incorrect attempt and using the difference between the two to generate feedback. In our case, the feedback is automated, giving the student a hint that they are close and what line number they could change to reach a solution. Such a system could also be used to reduce effort for a human grader. That means that instead of the vague problem of what the best message is to give a student who has not correctly solved an assignment, we instead only need to synthesize a modification to the student's attempt such that it matches the behavior of a reference solution.

To clarify what we mean by input in this context, we note that the input from the student is fixed: the system gets the student’s attempt and nothing else (except maybe the student’s past attempts on that or other assignments). On the other hand, the synthesizer may require some input from the creator of the assignment to perform well. In fact, Autograder does just this and requires an error model for each assignment describing how it should try mutating the student’s attempt to reach a correct program.

Our goal was to instead require the easiest input from the assignment creator possible: nothing at all other than a single reference solution.

To achieve this, we observed that the data available to the system includes not just the current student’s attempt but every other students’ attempts. That data provides a lot of information on what may or may not be useful for a solution. That data mining combined with the iterative synthesizer developed for the Test-Driven Synthesis project is able to synthesize solutions and thereby provide hints. As the model of finding a nearby solution applies only when the student is actually close to a solution, when the student is further away from a solution, we use that data to provide hints based on the information which is statistically most likely to lead to a solution.

We not only reduced user effort by not requiring an error model, but by using the domain-agnostic Test-Driven Synthesis technology, we developed a hint system that can work in any domain with no additional effort.

More information on the Code Hunt hint system can be found in Chapter 5.

1.5 Outline

Chapter 2 discusses background material relevant to the contributions of this work.

The next three chapters describe the projects developed with our input-driven design philosophy. Chapter 3 describes our API discovery system. Chapter 4 describes our end-user programming by example system. Chapter 5 describes our automated grading system.

Chapter 6 concludes and discusses future work.

Chapter 2

BACKGROUND

2.1 *Program synthesis*

Program synthesis covers a wide range of techniques for generating programs in variety of domains from a variety of inputs. For the purposes of this overview, we focus on the techniques used in our work and those which we compare against.

2.1.1 *Signature of a program synthesis algorithm*

A program synthesis algorithm outputs a program P given some description D of the program to be synthesized. The program P is in some language \mathcal{L} which depending on the algorithm may be a parameter or may be designed into the algorithm.

Exactly what information the algorithm needs about \mathcal{L} varies, although most need at least an interpreter or implementation of the functions in the language.

The description D , as discussed, may take many forms, but for our purposes it will always be a property required of the output program P that can be checked mechanically and therefore we can derive an oracle O from D for which the valid outputs of the algorithm are any $P \in \mathcal{L}$ such that $O(P)$ is true. The simplest example of this is test cases, which can easily be run against a candidate output program to verify it is in fact a valid output. Other descriptions like a reference program or a logical specification may be checked directly using a solver, but they may also be used to produce test cases. On the other hand, descriptions like a natural language sentence stating the desired behavior are out of scope.

2.1.2 Enumerative synthesis

The simplest program synthesis algorithm is basic enumerative synthesis [24]. Given a language to synthesize in and an oracle for verifying programs, the most basic enumerative synthesizer merely enumerates all programs in the language, checking each one against the oracle. The first program to satisfy the oracle is outputted.

Unsurprisingly, this brute force approach is quite inefficient and admits some simple optimizations [25, 26].

The first optimization is to avoid repetitive work by using dynamic programming. Record all program subtrees generated so far for each non-terminal in the grammar. When that non-terminal is reached again, instead of regenerating all program subtrees from scratch, just read off the list of those generated previously. In practice, as any non-trivial language \mathcal{L} will have a recursive grammar, this results in an algorithm that runs in stages. The first stage generates all of the subtrees of height 1. The n^{th} stage uses subtrees of height $n - 1$ in building the subtrees of height n .

Once all of the subtrees are being recorded, the next step is to detect which are semantically equivalent and keep only a single canonical subtree. As semantic equivalence of programs is not computable in general, each enumerative synthesis algorithm has different heuristics for detecting semantically equivalent programs.

One choice used by Katayama et al. [26] is to choose a list of parameter values they care about, and consider two subtrees distinct only if they are distinct for one of those parameter values. For example, $\mathbf{x+x}$ and $\mathbf{x*x}$ are clearly semantically distinct programs, but for the parameters values $x = 0$ and $x = 2$ they are indistinguishable as they both evaluate to 0 for $x = 0$ and 4 for $x = 2$. In Katayama et al.’s algorithm, they perform an iterative deepening search where they initially care about few parameter values and incrementally add more when they fail to find a solution. For instance, if they really did need to distinguish between $\mathbf{x+x}$ and $\mathbf{x*x}$, then any additional value for x like $x = 1$ would distinguish them. Implicit in this design choice is that all subtrees have values; in the referenced work, the target language

is Haskell, which is a functional programming language, so all subtrees of a function are expressions.

In addition to the simplicity of the algorithm, enumerative synthesis requires the minimal amount of information to be provided: the description is treated as a black box oracle and the language is only used for its semantics of how to execute the generated programs.

An enumerative solver is an undirected search method. It merely tries to find the shortest program and may have optimizations to avoid wasting time considering multiple programs with the same semantics, but due to treating the description and language both as black boxes, it is not a directed search in any sense. That suggests improvements involving some kind of directed search.

2.1.3 Hill-climbing synthesis

Hill climbing is a general term for search strategies that keep one or more current candidate answers and on each iteration try to move from those answers to one or more nearby candidate answers that are better. Exactly what nearby means and how those nearby values are discovered varies. The simplest form of a hill climbing search is to keep track of a single point in a space which is the domain of a real-valued objective function, and on each iteration compute the gradient of the objective function which will point in the uphill direction and update the point by moving some amount in that direction.

In terms of program synthesis, a hill-climbing algorithm keeps track of one or more best-so-far programs and at each iteration mutates those programs in attempt to generate better programs.

The catch is that this technique requires an objective function or some way to say one program is closer to the right answer than another. (We can avoid the need for computing or approximating a gradient in hill climbing by merely trying nearby objects blindly and checking if any of them are better according to the objective function.) In general, this is significantly harder to provide than the boolean oracle we assume.

Genetic programming

The most successful application of hill climbing to program synthesis is genetic programming [38]. In genetic programming, multiple candidate programs are maintained and at each step many new programs are generated by splicing together and mutating the known programs. Then each of these programs is tested against an objective function and only the highest scoring programs are kept for the next iteration.

Well-behaved objective functions can only be written for certain domains. One domain where they are easy to write is for approximations of expensive to compute functions. Then the objective value is just the error on some selected inputs. To avoid overfitting, the objective function can also include a term that discourages longer programs.

As one concrete example of a genetic programming system, ADATE [43] takes as input a set of test inputs and quality functions for their outputs and performs a genetic search where programs are mutated and only programs that improve or maintain the sum of the quality function values are kept.

Automated program repair

Automated program repair [62, 63, 32] takes a different perspective on the program synthesis problem, specifically that the description, along with a set of both passing and failing test cases, includes a program that is close to the right answer, so it is searching for a mutation which will make the program pass the failing test cases. The task is not viewed as synthesizing new functionality but instead as repairing functionality already present in the code. As a result, the existing code can be effectively used to guide many repairs, thereby avoiding many useless programs.

As the program is assumed to be mostly correct, automated program repair is can be done in two phases: fault localization and fault repair. That is, first the algorithm determines where to try mutating the existing program and then it attempts different mutations in those location. A common assumption in program repair is that the mutation can be pieced

together out of code appearing elsewhere in the same program.

Automated fault localization—determining which lines of a program may be buggy—can be used for manual debugging but can also be coupled with automated repairs.

The simplest automated fault localization technique is to look at the lines executed by the failing test cases: the bug must be in one of those lines. This can be refined by the concept of program slicing [64] wherein only the lines that actually affected the value under consideration are looked at, eliminating parts of the program logic which are irrelevant to the failure. There are many ways to define and compute slices [59].

Those techniques just look at which lines affected the output. We can further consider trying to determine which of those lines could actually be changed to fix the program. Bug-Assist [23] uses MaxSAT to select the smallest set of lines in a trace that could be changed to fix an error. Angelic debugging [7] tests candidate repair points by replacing them with symbolic values and uses symbolic execution to determine if there is any value that makes the test cases pass. This results in a conservative estimate of whether a repair is possible at that point because there is no guarantee that any known expression will compute the required values.

Stochastic search

Stochastic search methods sample the search space instead of exhaustively enumerating all possibilities. When the search space is relatively dense with solutions, stochastic methods can be faster at the cost of possibly failing to find the best solution.

One domain where stochastic search works particularly well is superoptimization, which performs the same task as normal compiler optimizations, but instead of using a predefined set of rules instead tries to find the absolute best equivalent code according to some performance metric [13, 37]. In the terms of the signature discussed above, D and P are both programs in \mathcal{L} , but P minimizes some cost function that approximates the performance cost of running the program on real hardware. Since for a superoptimization problem there are lots of answers, and finding an answer isn't hard (after all, the input D is a valid answer),

it’s okay to miss some answers—and give up guarantees of optimality—in exchange for being able to cover a larger search space.

Schkufza et al. [49] make that trade-off. The stochastic search maintains a set of candidate programs and scores them on both correctness and performance. The correctness criteria compares the bits of the output of the reference code and the synthesized code, scoring higher for more similar bits; the intuition for why this similarity tends to be meaningful in this domain is that bit twiddling tricks may result in partially computing the correct value. The performance criteria is an estimate of the processor time to execute the code which is empirically a good model of hardware execution times. To find new programs, the system samples mutations of the existing programs and keeps the results that score the best on those criteria. Note the difference from other algorithms that the mutations are sampled instead of exhaustively enumerated.

In practice, it turns out the algorithm given above does not actually produce superoptimized code; the results are similar to normal optimized code because it is overfitting to the original approach captured by the input program. The actual algorithm requires one tweak: an initial phase focused only on correctness, not scoring performance at all, which the authors call the synthesis phase to contrast it with the optimization phase. The synthesis phase generates multiple approaches to writing the program, which may be significantly slower than the original program, while the optimization phase finds the fastest variant of each of those approaches.

2.1.4 Solver powered methods

Another way to direct the search is use a SAT or SMT solver. One way to look at such tools is that they take a formula and do a search through all possible values of the variables in the formula for one that satisfies the formula. Given an encoding of the space of programs \mathcal{L} such that the variable values can be mapped to a program P and an encoding of the oracle O , a solver can be used as a synthesizer.

The actual approach taken by Sketch [52] and Rosette [60] is a bit different. Instead of

trying to phrase the synthesis problem as a single solver query, the solver is queried multiple times using an algorithm called CEGIS (CounterExample-Guided Inductive Synthesis). The key insight is that finding a program consistent with a small number of examples is easy, and the behavior of most (useful) programs can be described by how they act on a small number of general cases and an additional small number of edge cases. CEGIS splits the synthesis task into the two subtasks of generating test cases and synthesizing a program given a small set of test cases.

The first part takes a program and the problem description and outputs a counterexample, that is, a test case that the program gets the wrong answer on. If it can't find a counterexample, then the program is correct and the synthesis process is complete.

The other part takes a list of test cases and synthesizes a program that satisfies those test cases. If it can't find a program, then the synthesis process fails. Note that failing to find a program may involve multiple failed queries because the formula given to the solver has some finite size, so it only admits solutions up to some size bound; the query can be repeated with a larger size bound.

Rosette

Rosette [60] is a platform for building “solver-aided languages” for many use cases, including program synthesis. It is built on top of the Racket [1] dialect of Scheme, so Rosette can be used to synthesize any DSL that can be written in Racket. Such DSLs are not limited by language features which can be easily supported by an SMT solver because Rosette uses symbolic execution to bridge between the solver and the language runtime [61].

2.1.5 Version space algebras

All of the approaches discussed so far are “bottom-up” synthesis algorithms. They all work essentially by putting programs together from the pieces defined in \mathcal{L} and checking if they are correct, although exactly how that process is done varies widely. The opposite of this

is a “top-down” synthesis algorithm, which involves expanding the choices of the start non-terminal in \mathcal{L} and figuring out what the program could look like for each choice.

For example, let \mathcal{L} be a simple string manipulation language where a program consists of a concatenation of substrings of the input. Given the example $f(\text{"abc"})=\text{"ac"}$, the string "ac" could be the concatenation of $[\text{"ac"}]$, $[\text{"a"}, \text{"c"}]$, $[\text{"a"}, \text{"", \text{"c"}]$, or any of those with additional empty strings concatenated in. All of those are hypotheses for the input to the concatenation operation; we defined the arguments of the concatenation operation to be substring operations, so we would need to find ways to define substrings of "abc" to get $\text{"", \text{"a"}, \text{"c"},$ and "ac" . We can immediately see that "ac" is not a substring of "abc" , eliminating that possibility: therefore the program must concatenate more than one string together. One possible program is the concatenation of the substrings containing the first character and the third character of the input.

This intuition is formalized by version space algebras [40, 29], which allow for efficiently computing a succinct representation of the space of valid programs for a set of examples. Due to the support for efficient intersection, efficiently finding all programs for a set of examples follows directly from efficiently finding all programs for a single example.

In order to use a version space algebra, instead of needing implementations of the functions in \mathcal{L} , we need inverses which tell us which inputs could give a known output like how in the example we needed to know which lists of strings could be concatenated to form the string "ac" . This limits our language to functions that have inverses that are both easy to compute—no cryptographic hash functions allowed—and have small or compactly represented inverses. To clarify what is meant by the second requirement, consider the function $x+y$ over integers. No matter what it equals, there are as many choices for (x, y) as there are integers, each of them could take on any value, although the value of the other would depend on that choice. In practice this could be fixed by bounding the range of x and y , but it serves to demonstrate some care has to be taken when defining languages for use with this model.

Ranking

As version space algebras make finding all programs that satisfy a given example easy—and, in practice, the set is either empty or large—they lead to asking the follow-up question of which one to actually use. Hopefully, the program chosen will generalize to fresh examples the system hasn’t seen yet, so heuristics are chosen to maximise that chance.

As is common in learning problems, the principle of parsimony (Occam’s Razor) is followed: a smaller program is less likely to overfit. Unfortunately, that cannot be the singular overriding principle because given a single input/output example, the smallest program (assuming the DSL allows it) is to merely always return the output. Clearly this is unlikely to generalize, leading to a counterbalancing heuristic that computations are better than constants.

By computations are better than constants, we mean that if some subtree of the program can be computed based on the input, that is better than using a constant value. This maximizes the generalizability of the program at the cost of possibly overfitting due to spurious references to the input. As always, one way of dealing with overfitting is more examples, but a carefully designed system can often work with only a single example, so that is not enough. Domain knowledge can inform heuristics on which constants are more likely. For example, in string manipulation, punctuation is likely to be a constant while letters are much less likely to be constants [17]. More importantly, version space algebras are usually designed around fairly restrictive DSLs, so where constants are even allowed in the DSL is a carefully considered design choice.

2.1.6 Summary

We have classified program synthesis algorithms into four categories with different trade-offs. Enumerative synthesis optimizes for design simplicity and minimal burden in defining \mathcal{L} and D in exchange for only being able to cover a relatively small search space. Hill-climbing synthesis algorithms are useful when there is a well-defined notion of an almost correct

program. Solver-based techniques can accept any \mathcal{L} , albeit at the cost of design burden for the implementer of the algorithm. Version space algebras are highly efficient when they can be applied but require careful design of \mathcal{L} .

2.2 API discovery

Modern programming frameworks such as those found in Java and .NET consist of a huge number of classes organized into many namespaces. For example, the .NET Framework 4.0 has over 280,000 methods, 30,000 types, and 697 namespaces. Discovering the right method to achieve a particular task in this huge framework can feel like searching for a needle in a haystack. Programmers often perform searches through unfamiliar APIs using their IDE's code completion, for example Visual Studio's Intellisense, which requires the programmer to either provide a receiver or iterate through the possible receivers by brute force. Fundamentally, today's code completion tools still expect programmers to find the right method by name (something that implicitly assumes they will know the right name for the concept, which may not be true [11]) and to fill in all the arguments.

2.2.1 API discovery as search

There are two different approaches to this problem. The first is to use natural language search through documentation or programming help websites. This works well when the programmer knows what they want to do, can phrase it the same way others do, there is documentation to search (true for popular APIs, but might not be true for APIs which are less popular or not publicly available), and, perhaps most importantly, sees the problem as worth context switching from writing code to performing a search. There are projects which attempt to better integrate search into the IDE. For example, [6] performs a web search for a query and returns the highly ranked StackOverflow results.

Along those lines, but using documentation instead of web searches, SNIFF[8] returns snippets matching natural language queries by mining multiple examples from existing code, matching them based on the documentation of the APIs they use, and combining them

based on their similarities to eliminate the usage-specific parts of the snippets. Unlike most code completion algorithms, this technique requires the API being searched to be well-documented.

2.2.2 API discovery as synthesis

We are interested in the other approach, which is to view API discovery as a program synthesis problem. Or, in other words, as an extension to existing code completion mechanisms.

As a program synthesis problem, API discovery takes a description D consisting of the program written so far and the expression being completed and returns an ordered list of completions (which may just be the name of the next method to call, not an entire expression). As the description does not include anything as limiting as a logical specification or even test cases, there are usually a very large number of results, making ranking important to ensure the useful answers appear in the first few results. The other main design question is what information to expect the programmer to provide or to read from the context. For example, given the cursor position, the compiler can probably report the type required for an expression to be valid at that position; the synthesizer could use that information to only return expressions of that type.

In this model, the code completion provided by modern IDEs like Eclipse and Visual Studio takes as input just a type and generates snippets of just one additional method call, ranking the options alphabetically. This is still useful for API discovery because that list is often enriched with the full type signatures and documentation of those methods, allowing a programmer to browse through the available methods.

2.2.3 Jungloids

Prospector [35] specifically looks at the problem of finding a chain of API calls to convert from an input type to an output type, which the authors call a “jungloid”. In order to ensure the resulting code snippet is sensible, a precomputation step mines method calls, field lookups, and downcasts converting between various pairs of types from existing code.

Those snippets form the edges of a graph where the nodes are types. Then the synthesis problem is reduced to finding a path through that graph from the input type to the output type and merging together the snippets found along that path.

The motivating example in the Prospector paper is converting an `IFile` to an `ASTNode` in the Eclipse API which requires a non-obvious intermediate step involving a third type:

```
IFile file = ...;
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(file);
ASTNode ast = AST.parseCompilationUnit(cu, false);
```

Unlike most program synthesis algorithms, but common for code completion, Prospector includes a precomputation step. Code completion systems must deal with both the user expectation that queries will be answered very quickly (i.e., well under a second) and the fact that they are searching over large type hierarchies—the correspondence in other program synthesis systems would be large DSLs, which no system handles well in general.

The Prospector paper calls for using the principle of parsimony to rank results: the paper notes that shorter jungloids tend to be more likely to be correct. It also recommends the heuristic that jungloids that cross package boundaries are less likely to be correct.

PARSEWeb [54] performs the same task as Prospector except instead of using a fixed corpus (e.g., the code on the user’s computer) to build the graph, it searches for code examples using the input and output types from Google Code Search Engine searches and mines those to construct the graph. As such a snippet might not exist on the web, it has a fallback of trying to guess intermediate types in order to get enough code from the web searches to recover a subgraph that actually has a path from the input type to the output type.

As is common in the code completion domain, the set of all valid programs is not actually considered, in favor of only looking at programs that look at least somewhat similar to programs humans have actually written. This has a dual result of both greatly reducing the search space and filtering nonsense from the search results.

2.2.4 Exhaustive search

Typsy [5] searches for APIs by generating expressions involving any number of method and constructor calls and field lookups given a list of arguments, a return type, and a library package to search within. Typsy will only return expressions with all arguments filled in, so it will generate expressions to construct any missing arguments for methods it finds. The expressions are ranked by their size in order to avoid overly complicated results.

InSynth [19, 18] also produces expressions for a given point in code using the type as well as the context to build more complicated expressions. It generates expressions from scratch with no input from the programmer to guide it.

InSynth mines usage information from existing code so it can prefer more commonly used symbols. InSynth observes, as do many other code completion projects, that the context of a query provides a lot of information without the programmer having to explicitly include that information in the query. Symbols in the same methods are greatly preferred over symbols else in the file which in turn are greatly preferred over symbols imported from other files.

The weight function also implicitly encodes a parsimony assumption because the weights are summed together so larger expressions have a higher weight. That gives three criteria—small expressions using nearby symbols or commonly used imported symbols—which together are how InSynth is able to synthesize useful expressions.

2.2.5 Dynamic code completion

CodeHint [12] is a dynamic code completion method. By executing the code up to the point of the completion, CodeHint supports testing its generated expressions against user specified test cases in order to drastically reduce the number of the answers it returns. Additionally, the user can evaluate the options in terms of the values they compute. CodeHint generates expressions by a modified exhaustive search which looks at the frequencies of methods in actual code to determine how likely they are to be useful. Additionally, CodeHint allows for a skeleton of the code to be provided to narrow the search space to only the parts the

programmer isn't sure how to fill in.¹

2.2.6 Keyword programming

As a somewhat more search-like point of view on the program synthesis problem, Little and Miller[34] propose a system using “keyword programming” to generate method calls where the user gives keywords and the system generates a method call that includes arguments that have most or all of the keywords (or their synonyms). Their system attempts to be closer to natural language than ours at the cost of a lower success rate. API Explorer[10] also supports filtering results by a keyword, along with queries similar to the previously discussed systems.

This system generates code snippets from a very different kind of constraint than any other synthesis system considered so far, namely keywords that should appear in the synthesized program's symbols. In practice this turns out to be a poor form of description, at least when used in this way, despite it being a natural way for the user to frame a query.

2.2.7 API search using types

Another way to combine the search and synthesis views of the problem is to search for functions by their type signature. Searching for functions by type has been recommended for functional programming languages[48][66]. Those proposals differ in that the type signature alone, along with modifications to, for example, handle both curried and uncurried functions, tends to be sufficient for a search. In imperative languages with subtyping, inexact matches are more likely to be meaningful and side-effects make it more likely that many options have the same type.

For discovery of entire modules at once, specification matching can search by specification[67]. Semantics-based code search[47] similarly searches based on specifications including tests and

¹This is the partial expression concept we claim credit for in Chapter 3; the CodeHint paper cites our original publication [46].

keywords but additionally may make minor modifications to the code to fit the details of the specification.

2.2.8 Synthesizing glue code

MatchMaker[65] handles API discovery at a different scale: given two types, it generates the glue code to connect those two types by generalizing examples from existing code. The programmer only has to provide the two types and MatchMaker uses a precomputed dynamic analysis on typical executions of the library code to discover how the two types interact in order to drive the construction of the glue code. As the interactions in a dynamic trace will be overfitted to the specific uses in the trace, the algorithm strips away any method calls that do not cross the boundary between the user code and the library code.

2.3 Programming by example

Programming by example [16] and programming by demonstration [9] both refer to systems that allow non-programmers to automate repetitive tasks by performing one or more instances of the task and then asking the computer to learn what they are doing and repeat it. We distinguish the two concepts by saying that programming by demonstration systems take as input the entire trace of the user performing the action while programming by example systems only use the final state.

2.3.1 Advanced macro systems

The early programming by demonstration systems were essentially advanced macro systems [9, 33]. Instead of attempting to generate a program for an entire task, they infer a statement for each task, and show the program to the user in a user-friendly manner. For each domain, a new domain-specific language (DSL) is developed with an end-user-friendly syntax: the statements are English sentences with their editable arguments shown as drop-down or input boxes. Inferring statements is not always trivial; some of the system

descriptions mention using heuristics to choose when there are multiple options, but the lack of discussion of algorithms implies the search space is tiny by design. The idea is that a sufficiently expressive program for a specific instance should generalize to other instances simply by deleting the parts overly specific to the example instance. Either the system will properly guess where to generalize, or the program is shown to the user in a form that even a non-programmer can easily generalize it.

One important take-away, that recurs throughout program synthesis research, is that the selection of a good DSL is important. Here the balance is between it being high-level enough to be describing actions a non-programmer will understand while matching the program logic closely enough that there are few choices when inferring statements.

2.3.2 *Enumerative synthesis*

A DSL can be structured in such way that enumerating all of the possible programs is efficient. In Harris et al. [20], the task is to perform spreadsheet table transformations by example. By table transformations, they mean the output table only contains cells that exist in the input table, but in a different arrangement and possibly only a subset of them. The programs are made of mostly independent pieces that write out different subsets of the output which are overlaid to produce the entire output table. The mostly independent part is because a program fragment can place cells relative to those placed by another program fragment. The algorithm iterates over all possible base fragments and then recursively defines additional fragments based on any fragments that are found consistent with the output. The consistency requirement prunes the search space to make the search through all programs tractable. It is only due to the particular limited structure of the DSL that such a search can be made tractable.

The authors observe that in order to avoid overfitting, they follow Occam’s Razor and attempt to minimize the number of program fragments actually used in each program.

The authors explicitly punt on a complete solution to overfitting. Particularly, they suggest that while in theory it seems like it would be great if the program synthesizer could

come back and ask for clarification whenever it generates multiple different programs, there is the problem that the space of inputs is ill-defined and the user likely does not have a good concept of what it looks like other than simply pointing at the data the user wants to work with. The alternative they suggest is that in real use cases, the users should just feed examples to the system until they no longer need to correct its outputs, which will hopefully only take 1–2 examples.

2.3.3 Version space algebras

As previously discussed in Section 2.1.5, version space algebras are a data structure for programming by example which allow for efficiently computing a succinct representation of the space of valid programs for a set of examples [40, 29]. Version space algebras have been used for programming by example or demonstration in the domains of text editing [29], string manipulation [17], shell scripts [30], Python programs from traces [31], and repetitive robot programs [45].

2.4 Automated grading

Grading is a repetitive process that requires a lot of time from skilled teachers or teaching assistants. In introductory computer science classes, many of the assignments are code which can be easily verified correct by a computer by using a test suite, but assigning partial credit for incorrect solutions remains difficult. Program synthesis can partially automate this process. Additionally the same technologies can be used in an interactive tutoring system or educational game.

2.4.1 DFA constructions

Alur et al. restrict themselves to the easier problem of automated personalized feedback for finite automata [3]. Unlike general programs, many properties of finite automata are computable, but it turns out generating meaningful feedback is difficult. Their work is based

on modeling a few varieties of student errors in order to determine the best feedback to give, which they call problem syntactic mistakes, solution syntactic mistakes, and problem semantic mistakes.

A problem syntactic mistake means that the student solved the wrong problem and the system is able to tell them what problem they did solve, so the student can see that it is different. The paper introduces a DSL for DFA construction assignments, which gives small formulas for the languages that students are actually asked to write DFAs for. To determine what problem the student solved, a formula is synthesized in that DSL using the student program as the specification. Due to the assumption that useful formulas are small, the synthesis is done brute force via an iterative deepening search in perhaps the most extreme example of a well-chosen DSL making synthesis easier.

A solution syntactic mistake means a small correction to the student's code results in a correct solution, similar to the task Autograder performs. The space of changes explored is small enough that it suffices to exhaustively search for valid edits.

Lastly, a problem semantic mistake means that there are a small number of examples on which the code is incorrect, so the student can be shown one or more counterexamples.

Having all three kinds of feedback allows the system to be much more robust, as likely at least one of the three subsystems will identify a mistake with high confidence. Only the result from the subsystem that awards the highest grade is used, so it's okay if the other subsystems find a spurious similarity with a very different problem definition or solution DFA.

2.4.2 Autograder

AutoGrader [51] takes student programs and uses Sketch to search for similar programs that match the behavior of the reference solution. AutoGrader is presented on Python, but support for each programming language is a separate plugin. In order to apply AutoGrader to an assignment, the teacher must provide an error model which specifies the common errors that AutoGrader may correct. For example, the error model could say that for each

expression like $a+b$, to try $a-b$, $a*b$, and a/b .

This is a partially automated grading system because it does not assign scores, leaving that to a human grader. Also, by design, it only corrects solutions that are nearly correct; in practice, a human would still have to analyze student programs that are further away from a solution or have minor errors not representable as local changes.

Restricting the error model to a well-defined set of changes ensures that the changes will not be too surprising as opposed to allowing arbitrary mutations of the student code which could result in code which is not similar in any meaningful way, which might be no easier for a human to understand than the student's original error.

2.4.3 Non-synthesis approaches

Moss

Moss [50] automates plagiarism detection for programming assignments by finding similar programs and suggesting to a human grader that the similarity may be evidence of plagiarism. While not a grading mechanism, plagiarism detection is a task that human graders often want to perform but can be performed much better by a computer. Also, the general concept of looking for similar solutions is used to get more information in the CodeWebs project discussed below.

Code Hunt

Code Hunt [56] (and its predecessor Pex4Fun [55]) provide automated feedback in the form of counterexamples. Both use the same central game concept of having the student write a program with little or, most often, no specification. For each attempt submitted, the student is shown test cases which include the answer given by the secret reference solution. After enough attempts, the test cases eventually give the student enough information to determine the specification and write a correct program.

By using the Pex [57] test case generation tool, Code Hunt and Pex4Fun are able to

generate counterexamples for any incorrect student program given only a reference solution. This both avoids the work of producing a test suite by hand and the problems of the test suite being incomplete or discovered by the student. That last case would be a problem for a game like Code Hunt where the player can make an unlimited number of attempts.

As discussed above, test cases alone are not always sufficient feedback, and they are a poor measure of how close to a solution a student actually is.

CodeWebs

CodeWebs [41] takes a large number of submissions along with the results of a test suite on each one and groups them by syntactic similarity. By finding correct programs that are almost identical except for one subexpression replaced with another, it can determine that, for the purposes of that problem, those two subexpressions are equivalent, and by further application of that logic build equivalence classes of expressions. That data can be further used to localize bugs by identifying expressions that appear only in incorrect code, and even recommend fixes by finding a similar correct solution in the data.

Machine learning

Varun et al. [53] perform automatic grading by doing machine learning looking at large number of features based on counts of expressions, expressions in a context (like an `if` statement or a nested loop), and data dependencies like a comparison in a nested loop depending on a post-increment in a loop. All of these counts make a feature vector for a machine learning algorithm. The algorithm is trained with submissions of known scores and then is able to estimate the scores of new submissions based on what features they have. The features can be complicated because they do not appear in feedback shown to the student, so they do not have to be human-readable.

2.5 *Lessons learned*

In summary, the following common threads were seen in many projects, which should inform any program synthesis project:

1. The principle of parsimony helps avoid poor solutions, preventing both overfitting and complexity.
2. Choosing a good DSL is important, often in unintuitive ways like the choice of where in a DSL to allow computed as opposed to constant values. In particular, the design decisions may require both expert knowledge of the domain and expert knowledge of the synthesis algorithm. Hopefully any general purpose synthesizer minimizes the need for the author of the DSL to be knowledgeable about the inner workings of the synthesizer.
3. When working with incomplete specifications, some amount of overfitting is unavoidable because for overfitting to be a well-defined concept, there must be an idea of what the space of inputs is, which the user is unlikely to have in mind very precisely. The input space for a string manipulation program isn't the space of all strings, it's the much more vaguely defined space of all strings the user cares about.
4. The user interaction model should inform the synthesis strategy. The advanced macro systems could use simpler algorithms due to the assumption that the users would be willing to read and modify the synthesized code. The table transformations system did not bother with a complex way to ensure the user got what they wanted; they just assume the user will invoke the system again if necessary.
5. The domain can inform very different ways to avoid poor solutions. In much of the code completion work, the starting point for synthesis is some piece of user code as opposed to just the definitions of the functions being composed. On the other hand,

in superoptimization, generating poor solutions turned out to be a necessary starting point.

Chapter 3

PARTIAL EXPRESSION COMPLETER

We defined the API discovery problem in Section 2.2 as a program synthesis problem where the description is the cursor position where the programmer wishes to write a new code snippet, possibly along with a small amount of additional information like a keyword or an object to use in the computation. We are particularly interested in scenarios where the programmer has limited familiarity with the library being searched.

Following our design philosophy of getting as much information out of the user as can be provided with minimal effort, we propose instead taking as the description a *partial expression*, by which we mean an expression in the programming language that is allowed to be incomplete in various ways. Thereby everything the user knows about the desired expression can be provided in a syntax as close to the actual programming language syntax as possible while leaving the unknown parts blank to be filled in by our algorithm.

We define a language of partial expressions in which programmers can indicate in a superset of the language’s concrete syntax that certain subexpressions need to be filled in or possibly reordered. We interpret a partial expression as a query that returns a ranked list of well-typed completions, where each completion is a synthesized small code snippet. This model is simple, general, and precisely specified, allowing for a variety of uses and extensions. We developed an efficient algorithm for generating completions of a partial expression. We also developed a ranking scheme primarily based on (sub)typing information to prefer more precise expressions (e.g., a method taking `AVerySpecificType` rather than `Object`).

By using a superset of the host language as the interface for queries, we simultaneously make even complex queries easy for programmer to understand and make it easy for the programmer to provide hints that significantly reduce the search space.

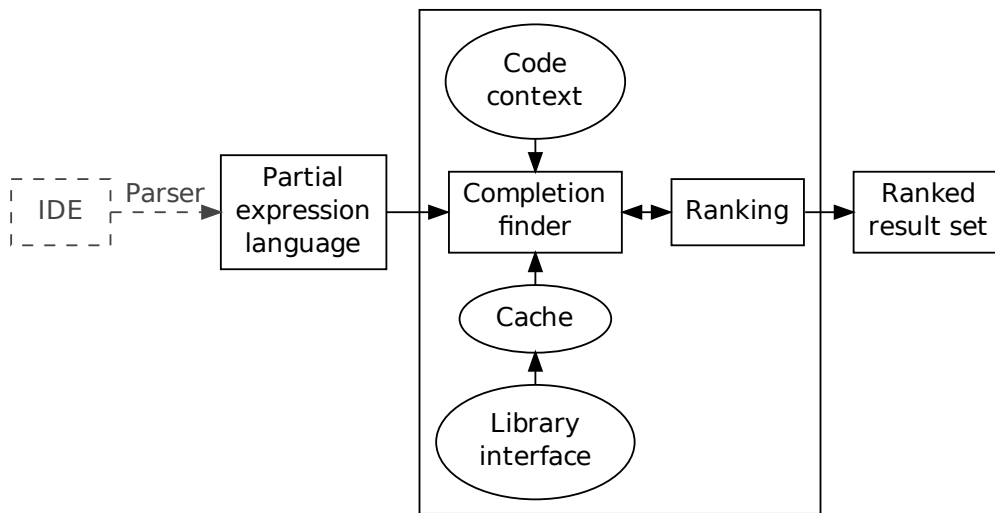


Figure 3.1: Workflow

Outline

First, we motivate the problem we are solving by capturing additional information (Section 3.1). Then, we define precisely what form partial expressions take (Section 3.2), give a ranking heuristic (Section 3.3) and algorithm (Section 3.4) for code completion given a partial expression, and perform a simulated evaluation of our ranking heuristic on real code in order to argue that easy to write partial expressions capture enough information for an effective API discovery system (Section 3.5).

3.1 Illustrative examples

This section describes three examples that use partial expressions in our system and then considers performing the same tasks using prior work. Section 3.1.1 describes how our system handles these examples. Section 3.1.2 discusses normal code completion. Section 3.1.3 compares to other research into API discovery like Prospector [35] and InSynth [19, 18].

Query: <code>?({img, size})</code>
Results:
<code>PaintDotNet.Actions.CanvasSizeAction.ResizeDocument(img, size, ◇, ◇)</code>
<code>PaintDotNet.Functional.Func.Bind(◇, size, img)</code>
<code>PaintDotNet.Pair.Create(size, img)</code>
<code>PaintDotNet.Quadruple.Create(size, img, ◇, ◇)</code>
<code>PaintDotNet.Triple.Create(size, img, ◇)</code>
<code>PaintDotNet.PropertySystem</code>
<code> .StaticListChoiceProperty.CreateForEnum(img, size, ◇)</code>
<code>System.Drawing.Size.Equals(size, img)</code>
<code>System.Object.ReferenceEquals(size, img)</code>
<code>PaintDotNet.Document.OnDeserialization(img, size)</code>
<code>PaintDotNet.PropertySystem.Property.Create(◇, size, img)</code>

Figure 3.2: The top 10 results generated by our system for `?({img, size})`.

3.1.1 Our system

Synthesizing Method Names

Suppose you are writing code using an image editing API (specifically, the Paint.NET image editor¹) and want to figure out how to make an image smaller. Your first instinct may be to write `img.Shrink(size)`. Unfortunately, that API does not exist; the actual API for shrinking an image is a static method under `PaintDotNet.Actions.CanvasSizeAction`:

```
public static Document ResizeDocument(Document document,
    Size newSize, AnchorEdge edge, ColorBgra background)
```

We will step through how our tool handles this example, using Figure 3.1, which shows the workflow of our tool. For this example, you would write the query “`?({img, size})`” in the partial expression language described in Section 3.2. The query is passed to the algorithm represented by the large box described in Section 3.4 which also has access to the code context which says that `img` and `size` are local variables of types `Document` and `Size`, respectively. The first ten elements of the ranked result set are shown in Figure 3.2. The

¹<http://www.getpaint.net/>

Query: <code>DynamicGeometry.Math.Distance(point, ?)</code>
Results:
<code>point</code>
<code>this.BeginLocation</code>
<code>this.Center</code>
<code>this.EndLocation</code>
<code>DynamicGeometry.Math.InfinitePoint</code>
<code>shapeStyle.GetSampleGlyph().RenderTransformOrigin</code>
<code>this.shape.RenderTransformOrigin</code>
<code>this.ArcShape.Point</code>
<code>this.Figure.StartPoint</code>
<code>this.Shape.RenderTransformOrigin</code>

Figure 3.3: The top 10 results generated by our system for the `?` in `Distance(point, ?)`.

static `ResizeDocument` method is the first choice.

In this example, the information the programmer provides to the tool is that some method should be called and it should have `img` and `size` as two of its arguments, although they may not be the only arguments and they may appear in any order. This captures the fact that in this example the programmer knows they are performing a computation so there will be a method call involved, but without familiarity with the library, the expectation of instance instead of static for a given method is just a guess. Due to not thinking through the operation as much as the library designer had, it may not occur to the programmer that the other arguments are necessary to clarify exactly how the resize should operate.

Our system does not attempt to fill in those extra arguments; \diamond is used to mark argument positions that the tool returns blank. The programmer could run additional queries to fill them in if necessary. The next example will be such a query.

Synthesizing Method Arguments

Suppose you already know there is a method `Distance(·, ·)` that returns the distance between two `Point` objects, but are not sure where one of the endpoints is defined. The query “`Distance(point, ?)`” produces a list of `Points` that could be filled in as the second

Query: <code>point.* >= this.*</code>
Results:
<code>point.X >= this.P1.X</code>
<code>point.X >= this.P2.X</code>
<code>point.X >= this.Midpoint.X</code>
<code>point.X >= this.FirstValidValue().X</code>
<code>point.Y >= this.P1.Y</code>
<code>point.Y >= this.P2.Y</code>
<code>point.Y >= this.Midpoint.Y</code>
<code>point.Y >= this.FirstValidValue().Y</code>
<code>point.X >= this.Length</code>
<code>point.Y >= this.Length</code>

Figure 3.4: The top 10 results generated by our system for `point.* >= this.*`.

argument. This includes any locals, fields, or static fields or methods or recursively any fields of those of type `Point`. For example, Figure 3.3 shows the results of that query in the context of the `EllipseArc` class of the `DynamicGeometry` library. In this case, the actual argument was `this.Center`, which appears third in the list.

This is the minimal amount of information the system can be given: just a cursor position. This is the degenerate case of partial expressions, which is equivalent to the queries supported by the related work.

Synthesizing Field Lookups

For a more targeted version of the above, the search can be narrowed by specifying the base object to look under. We will consider synthesizing field lookups in the context of a comparison operator. The query “`point.* >= this.*`” includes `point` and `this` along with zero or more field lookups or zero-argument instance method calls after them. The top ten ranked completions for this query are listed in Figure 3.4. Note that by completing both holes simultaneously, only completions where the two sides have fields of compatible types are shown.

3.1.2 Code Completion

Today, programmers can use code completion such as Intellisense in Visual Studio or the equivalent feature in Eclipse and other IDEs to try to navigate unfamiliar APIs. Intellisense completes code in sections separated by periods (“.”) by using the type of the expression to the left of the period and textually searching through the list for any string the programmer types. If there is no period, then Intellisense will list the available local variables, types, and namespaces. The options are listed in alphabetical order. This often works well, particularly when the programmer has a good idea of where the API they want is or if there are relatively few choices. On the other hand, it performs poorly on our examples.

Synthesizing Method Names

Using Intellisense to attempt to find the nonexistent **Shrink** method, a programmer might type “`img.shr`”, see that there is no “**Shrink**” method, and then skim through the rest of the instance methods. As that will also fail to find the desired method, the programmer might continue by typing in “`PaintDotNet.`” and use Intellisense to browse the available static methods, eventually finding `PaintDotNet.Actions.CanvasSizeAction` where the method is located. Hopefully, documentation on the various classes and namespaces shown by Intellisense’s tooltip will help guide the programmer to the desired method, but this is dependent on the API designer documenting the library code well and the documentation using terminology and abstractions that the programmer understands. A programmer would likely search through many namespaces and classes before happening upon the right one.

Synthesizing Method Arguments

If the user has already entered “`Distance(point,` ” and then triggers Intellisense, Intellisense will list every namespace, type, variable, and instance method in context even though many choices will not type-check (as the programmer may intend to call a method or perform a property lookup on one of those objects). When the list is brought up, the most recently

used local variable of type `Point`, which would be `point` in this case, will be selected. The programmer will have to read through many unrelated options to locate the other values of type `Point`.

Note that Eclipse’s code completion is actually significantly different in this scenario. It will list all of the local variables valid for the argument position along with common constants like `null`. If a more complicated expression is desired, the user has to cancel out and request the normal code completion which is similar to Visual Studio’s.

Synthesizing Field Lookups

Given “`point.`”, Intellisense will list all fields and methods of that object. The listing will go only one level deep: if the user wants a field of a field, they have to know which field to select first or browse through all of them. Furthermore, due to neither being type-directed nor completing both holes at once, Intellisense provides no aid in finding pairs of values that are comparable.

3.1.3 Other research tools

We consider two forms of input used by related work:

1. Source and destination types: In the language of partial expressions, this means giving a return type and specifying a single argument value that may have any expression wrapped around it to compute a value of that return type. This is the input form used by Prospector [35] and PARSEWeb [54].
2. Just a cursor position: This is exactly the same input as the partial expression `?`, which means to build any expression that matches the type required at the cursor position. InSynth [19, 18] and Typsy [5] take input of this form.

Synthesizing Method Names

This query does not fit into the source and destination types model. If we try to force it, then we could consider it as a conversion from `Document` to `Document`, but that does not capture the programmer’s intuition of wanting to resize the document and may match too many other options. Prospector will return methods with arguments it cannot fill in, but it prefers fewer unknown arguments, so `ResizeDocument` would likely be rather far down in the list of options for either query.

Given just a cursor position, the hint that the `img` and `size` variables are recent in the method body might be enough of a hint to put the desired method relatively high in the list of options, but it doesn’t give the programmer the opportunity to just tell the system what variables they want to use.

Synthesizing Method Arguments

As this query involved only a return type, as already mentioned, it does not really differ from the expressiveness available in queries to in the related work, so we would expect similar results.

Synthesizing Field Lookups

The related work doesn’t have a concept of multiple holes, so the two sides of the `>=` operator would have to be handled separately, losing the information that the types have to be comparable, and also requiring the user to select a type in order to make the query.

3.2 Partial expression language

Queries in our system are partial expressions. A partial expression is similar to a normal (or “complete”) expression except some information may be omitted or reordered. A partial expression can have many possible completions formed by filling in the holes and reordering subexpressions in different ways.

$$\begin{aligned}
\text{(a)} \quad e &::= call \mid varName \mid e.fieldName \mid e:=e \mid e<e \\
call &::= methodName(e_1, \dots, e_n) \\
\text{(b)} \quad \tilde{e} &::= \tilde{a} \mid \underline{?} \mid \diamond \\
\tilde{a} &::= e \mid \tilde{a}.\underline{?} \mid \tilde{a}.\underline{*} \mid \widetilde{call} \mid \tilde{e}:=\tilde{e} \mid \tilde{e}<\tilde{e} \\
\widetilde{call} &::= \underline{?}(\{\tilde{e}_1, \dots, \tilde{e}_n\}) \mid methodName(\tilde{e}_1, \dots, \tilde{e}_n)
\end{aligned}$$

Figure 3.5: (a) Expression language (b) Partial expression language

3.2.1 Complete expression syntax

Since partial expressions are a superset of some specific programming language, in order to formalize partial expressions, we need a programming language to formalize them on top of. To this end, we first define a simple expression language given by the e and $call$ productions in Figure 3.5(a), which includes relevant features found in traditional programming languages. Our simple language has variables, field lookups, assignments, a comparison operator, and method calls. (Other operators are omitted from the formalism.) Also, the receiver of a method call is considered to be its first argument in order to simplify notation as when reordering arguments, an argument other than the first may be chosen as the receiver. Adapting this definition to a specific real programming language should be straightforward.

3.2.2 Partial expression syntax

Partial expressions are defined by the \tilde{e} , \tilde{a} , and \widetilde{call} productions in Figure 3.5(b). Partial expressions support omitting the following classes of unknown information:

- **Entire subexpressions.** $\underline{?}$ gives no information about the structure of the expression, only that it is missing and should be filled in and its location may give a type for the expression. On the other hand, \diamond should not be filled in: it indicates a subexpression to ignore due to being independent of the current query (so making it a $\underline{?}$ would only add irrelevant results) or simply being a subexpression the programmer intends to fill

in later, perhaps due to working left-to-right.

- **Field lookups and simple method calls.** The \tilde{a} production defines the $\underline{.?}$ and $\underline{.*}$ suffixes which are slightly different ways of saying that an expression is missing one or more field (or property) lookups or the desired expression is actually the result of a method call on the expression. We do not distinguish between field lookups, property lookups, and methods that take only one argument like $\underline{.GetX()}$ as they tend to serve similar purposes. Naturally, different choices could be made to allow the programmer to express more distinctions if there were reason to believe they would be useful.

For zero or one lookup, $\underline{.?}$ is used, while $\underline{.*}$ completes as any number of lookups.

- **Unknown method name or arguments.** $\underline{?}(\{\tilde{e}_1, \tilde{e}_2\})$ represents a call to some unknown method with two known arguments, which may be partial expressions.

For unknown methods, there may also be additional arguments missing or the arguments may be out of order, which is represented by the use of set notation for the arguments in $\underline{?}(\{\tilde{e}_1, \tilde{e}_2\})$.

3.2.3 Partial expression semantics

Figure 3.6 gives the full semantics of the partial expression language. The \Downarrow judgement nondeterministically takes a partial expression to a complete expression with the exception that any \diamond subexpressions remain. With the exception of the $\underline{.*}$ rule, each rule removes or refines some hole, making the partial expression one step closer to a complete expression. The bottom rule allows for the composition of other rules. The top leftmost rule allows a $\underline{.?}$ suffix to be omitted. For type checking, \diamond is treated as a wildcard: as long as some choice of type for the \diamond works, the expression is considered to type check. The actual algorithm implemented does not use these rules exactly, although it matches their semantics.

The partial expressions language semantics never add operations like multiplication or new method calls (other than to zero-argument methods). The idea is that any place where

$$\begin{array}{c}
\frac{\tilde{e} \Downarrow e}{\tilde{e}.\underline{?} \Downarrow e} \quad \frac{\tilde{e} \Downarrow e}{\tilde{e}.\underline{?} \Downarrow e.m()} \quad \frac{\tilde{e} \Downarrow e}{\tilde{e}.\underline{?} \Downarrow e.f} \quad \frac{}{\tilde{e}.\underline{*} \Downarrow \tilde{e}.\underline{?}.\underline{*}} \\
\frac{\tilde{e}_1 \Downarrow e_1 \quad \tilde{e}_2 \Downarrow e_2}{\tilde{e}_1 := \tilde{e}_2 \Downarrow e_1 := e_2} \quad \frac{\tilde{e}_1 \Downarrow e_1 \quad \tilde{e}_2 \Downarrow e_2}{\tilde{e}_1 < \tilde{e}_2 \Downarrow e_1 < e_2} \\
\frac{\tilde{e}_i \Downarrow e_i}{m(\tilde{e}_1, \dots, \tilde{e}_i, \dots, \tilde{e}_n) \Downarrow m(\tilde{e}_1, \dots, e_i, \dots, \tilde{e}_n)} \\
\frac{\tilde{e}_i \Downarrow e_i}{\underline{?}(\{\tilde{e}_1, \dots, \tilde{e}_i, \dots, \tilde{e}_n\}) \Downarrow \underline{?}(\{\tilde{e}_1, \dots, e_i, \dots, \tilde{e}_n\})} \\
\frac{k \geq n, e_j = \diamond \text{ for } j > n \quad \sigma \in S_k}{\underline{?}(\{e_1, \dots, e_n\}) \Downarrow m(e_{\sigma_1}, \dots, e_{\sigma_k})} \quad \frac{v \text{ is a live local or global variable}}{\underline{?} \Downarrow v.\underline{*}} \\
\frac{\tilde{e}_1 \Downarrow \tilde{e}_2 \quad \tilde{e}_2 \Downarrow \tilde{e}_3}{\tilde{e}_1 \Downarrow \tilde{e}_3}
\end{array}$$

Figure 3.6: Semantics of partial expressions. The final result must type-check in the context of the query, treating \diamond as having any type.

computation is intended should be explicitly specified, and the completions simply list specific APIs for the computations. The exception for zero-argument methods is made because they are often used in place of properties for style reasons or due to limitations of the underlying language.

3.2.4 Examples

The first example from Section 3.1, $\underline{?}(\{\text{img}, \text{size}\})$, is a method call with an unknown name and two complete expressions as arguments. It can be expanded to any method that can take those two variables in any two of its argument positions, so `Triple.Create(\diamond , size, img)` is a valid completion. Note that we mark the extra argument space with \diamond to clarify that no attempt is made to fill it in. This is done to reduce the number of choices when recommending methods; for other applications fully completing the expression may be useful. The user may afterward decide to convert the \diamond to $\underline{?}$ or some other partial expression.

Our second example from Section 3.1, `Distance(point, $\underline{?}$)` can take one step to one of

- `Distance(point, point.*)`,
- `Distance(point, this.*)`,
- `Distance(point, shapeStyle.*)`

or many other possibilities. Any local in scope or global (static field or zero-argument static method) could be chosen to appear before the `.*`. Whatever is selected is completed to some expression of type `Point`. The `.?` suffix can be omitted when completing an expression, so `point.*` can be completed as `point` which is the first option in Figure 3.3. `this.*` can also become one or more lookups by going to `this.?.*` in one step and the `.*` becomes some field. For `ArcShape`, `this.ArcShape.*` is further completed to `this.ArcShape.Point`. So far we have only completed using fields and properties. On the other hand, for `shapeStyle.*`, the first `.?` from the `.*` is completed with an instance method `.GetSampleGlyph()` that returns an object with a field `RenderTransformOrigin` of type `Point` which the remaining `.*` can complete to.

An unknown method's arguments may themselves be partial expressions. For example, `?({strBuilder.*, e.*})` could expand to `Append(strBuilder, e.StackTrace)` (which would normally be written as `strBuilder.Append(e.StackTrace)`).

The third example from Section 3.1, `point.* >= this.*`, also uses `.*`, so the completions work as above, but, as there are two of them in the expression related by the `>=`, there must be a definition of `>=` which is type compatible with the two completions. In this example, all the comparable fields have types `int` or `double`. But suppose `Point` had a field `Timestamp` of type `DateTime`; then `Point.Timestamp >= this.P1.Timestamp` would be a valid completion, but `Point.X >= this.P1.Timestamp` would not as a `double` cannot be compared to a `DateTime`.

3.3 Ranking

Partial expressions often have a large number of potential completions, many of which, like the `Triple.Create(◇, size, img)` example in the previous section, are obviously unlikely

$$\begin{aligned}
\text{score}(expr) &= \sum_{s \in \text{subexprs}(expr)} \text{score}(s) \\
&+ \sum_{s \in \text{subexprs}(expr)} \text{td}(\text{type}(s), \text{type}(\text{param}(s))) \\
&+ 2 \cdot \text{dots}(expr) \\
&+ \sum_{s \in \text{subexprs}(expr)} \text{abstype}(s) \neq \text{abstype}(\text{param}(s)) \\
\\
\text{scorec}(call) &= \text{score}(call) \\
&+ (\text{isInstance}(call) \vee \text{isNonLocalStatic}(call)) \\
&+ \max\left(0, 3 - (\text{nsArgs}(call) \neq \text{reciever}(call)) \cdot \left| \bigcap_{s \in \text{nsArgs}(call)} \text{ns}(\text{type}(a)) \right| \right) \\
\\
\text{scorecmp}(expr_1, expr_2) &= \text{score}(expr_1 < expr_2) + 3 \cdot (\text{name}(expr_1) \neq \text{name}(expr_2)) \\
\\
\text{nsArgs}(call) &= \{a \in \text{subexprs}(call) \mid \text{type}(a) \text{ is not primitive}\}
\end{aligned}$$

Figure 3.7: The ranking function. Note that boolean values are considered 1 if true and 0 if false and that abstract types ($\text{abstype}(\cdot)$) are considered not equal if both are *undefined*.

to be useful. To address that, an effective API discovery algorithm must include a good way to rank the available completions.

The ranking function maps completed expressions that may contain \diamond subexpressions to integer scores. This function is used to rank the results returned by the completion finder in ascending order of the ranking score (i.e., a lower score is better). The function is defined such that each term is non-negative, so if any subset of the terms are known, their sum is a lower bound on the ranking score and can be used to prune the search space.

The computation is a sum of multiple terms summarized in Figure 3.7. $\text{score}(\cdot)$ applies to all expressions while $\text{scorec}(\cdot)$ is a specialized version for method calls and $\text{scorecmp}(\cdot, \cdot)$ is a specialized version for comparisons. The computation is defined recursively, so for methods or operators with arguments, the sum of the scores of their arguments is added to the score. The scoring function incorporates several features we designed based on studying code examples and our own intuition. This section explains these features in detail. Section 3.5.5 evaluates each feature’s contribution to our empirical results.

3.3.1 Type distance

The primary feature in the ranking function is “type distance”, for example from a method call argument’s type to the type of the corresponding method parameter. Informally, it is the distance in the class hierarchy, extended to consider primitive types, interfaces, etc. For example, if `Rectangle` extends `Shape` which extends `Object`, $\text{td}(\text{Rectangle}, \text{Shape}) = 1$ and $\text{td}(\text{Rectangle}, \text{Object}) = 2$. Qualitatively, a user is more likely to think of a `Rectangle` as a `Shape` than as an `Object`, so the ranking reflects that. Far away types are less likely to be used for each other, so method calls and binary operations where the arguments have a higher type distance are less likely to be what the user wanted.

Formally, the type distance from a type α usable in a position of type β to that type β , $\text{td}(\alpha, \beta)$, is defined as follows:

$$\text{td}(\alpha, \beta) = \begin{cases} \textit{undefined} & \text{no implicit conversion of } \alpha \text{ to } \beta \\ 0 & \text{if } \alpha = \beta \\ 1 & \text{if } \alpha \text{ and } \beta \text{ are primitive types} \\ 1 + \text{td}(\text{s}(\alpha), \beta) & \text{otherwise} \end{cases}$$

$\text{s}(\alpha)$ is the explicitly declared immediate supertype of α which minimizes $\text{td}(\text{s}(\alpha), \beta)$. Note that $\text{td}(\alpha, \beta)$ is used by the ranking function only when it is defined as that corresponds to the expression being type correct.

The type distance term is the sum of the type distances from the type of each argument arg to the type of the corresponding formal parameter $\text{param}(arg)$. For method calls this is well-defined; binary operators are treated as methods with two parameters both of the more general type, so the type distance between the two arguments to the operator is used.

3.3.2 *Depth*

The next term prefers expressions with fewer subexpressions. The ranking scheme prefers shorter expressions by computing the complexity of the expression which is approximated by the number of dots in the expression and multiplying that value by 2 to weight it more heavily. For example, `dots("a.foo") = 1` so it would get a cost of 2 while `dots("a.bar.ToBaz()") = 2` so it would get a cost of 4. To avoid double counting, any dots which are part of subexpressions are not counted here and instead are included via the subexpression's score term.

3.3.3 *In-scope static methods*

Instance method calls will tend to have a type distance of zero for the receiver, so type distance has an implicit bias against static method calls. Noting that static methods of the enclosing type can be called without qualification, just like instance methods with `this` as the receiver, our ranking algorithm should similarly not disfavor such in-scope static methods. This is fixed by adding a cost of 1 if either the method is an instance method or the method is a static method that is not in scope.

3.3.4 *Common namespace*

As related APIs tend to be grouped into nearby namespaces, the algorithm prefers calls where the types of all the arguments with non-primitive types and the class containing the method definition are all in the same namespace. Primitive types, including `string`, are ignored in this step because they are used with varying semantics in many different libraries. Furthermore, deeper namespaces tend to be more precise so a deep common namespace indicates the method is more likely to be related to all of the provided arguments. Specifically, the algorithm takes the set of all namespaces of non-primitive types among the arguments, treats them as lists of strings (so `"System.Collections"` is `["System", "Collections"]`), finds the (often empty) common prefix, and uses its length to compute the "namespace score". To avoid this boosting the scores of instance calls with only one non-primitive argument, the

similarity score is 0 in that case.

In order to have the namespace similarity term be non-negative, namespace similarities are capped at 3, and 3 minus the length of the common prefix is used as the common namespace term.

3.3.5 *Same name*

Comparisons are often made between corresponding fields of different objects. Whether two fields have corresponding meanings can be approximated by checking if they have the same name. That is, `p.X` is more likely to be compared to `this.Center.X` than to `this.Center.Y`. To capture this, a 3 point penalty is assigned to comparisons where the last lookups on the two sides do not have the same name. The value is intentionally chosen to be greater than the cost of a lookup, so a slightly longer expression that ends with a field of the right name is considered better than a shorter one that does not.

3.3.6 *Abstract type inference*

As a refinement to the type distance based on declared types already mentioned, we additionally do type distance on automatically generated “abstract types” based on usage, which is particularly important for commonly used types like `string`. Abstract types may have richer semantics than `string` such as “path” or “font family name”. Our approach is based on the Lackwit tool that infers abstract types of integers in C [42].

Abstract types are computed automatically using type inference. An abstract type variable is assigned to every local variable, formal parameter, and formal return type, and a type equality constraint is added whenever a value is assigned or used as a method call argument. As all constraints are equality on atoms, the standard unification algorithm can be implemented using union-find.

In order to avoid merging every abstract type `.ToString()` or `.GetHashCode()` is called on, methods defined on `Object` are treated as being distinct methods for every type. All other methods have formal parameter and formal return type terms associated with their

definition which are shared with any overriding methods. A more principled approach might involve a concept of subtyping for abstract types, but that would greatly complicate the algorithm for what would likely be minimal gain.

For example, consider the following code from `Family.Show`:²

```
string appLocation = Path.Combine(
    Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments),
    App.ApplicationFolderName);
if (!Directory.Exists(appLocation))
    Directory.CreateDirectory(appLocation);
return Path.Combine(appLocation, Const.DataFileName);
```

`Directory.Exists`, `Directory.CreateDirectory`, and `Path.Combine` all have the same first argument, `appLocation`, so the analysis concludes their first arguments are all the same abstract type. Furthermore, from the first statement, that must also be the abstract type of the return values of `Path.Combine` and `Environment.GetFolderPath`. On the other hand, there is no evidence that would lead the analysis to believe the second argument of `Path.Combine` is of that abstract type, instead `App.ApplicationFolderName` and `Const.DataFileName` are believed to both be of some other type. Intuitively, a programmer might call those two types “directory name” and “file name”, but for the purposes of the algorithm all that matters is identifying them as being distinct.

In the ranking function, the type distance computation is refined by adding an additional cost of 1 if the abstract types do not match.

Algorithm 3.1: AllCompletions(\tilde{e})

```

input  :  $\tilde{e}$ : a partial expression
output a generator of all completions in order by score
:
1 subexps  $\leftarrow$  the list of immediate subexpressions of  $\tilde{e}$ ;
2 Let subcomps be a map from subexps to completions;
3 foreach  $s \leftarrow$  subexps do
4    $\lfloor$  subcomps[ $s$ ]  $\leftarrow$  AllCompletions( $s$ );
5 foreach  $score \leftarrow$   $[0, \infty)$  do
6    $\lfloor$  foreach concreteSubs  $\leftarrow$  all choices of exactly one completion for each
7     subexpression from subcomps whose score(concreteSubs)  $\leq$  score do
8      $\lfloor$  foreach type-correct completion  $c$  of  $\tilde{e}$  using subexpressions concreteSubs
9       where score( $c$ ) = score do
10       $\lfloor$  yield return  $c$ ;

```

3.4 Algorithm

This section describes an algorithm (represented by the boxed section of Figure 3.1) for completing partial expressions. The algorithm takes a partial expression as input and returns a generator that returns the completions in ranked order one at a time. The algorithm has access to static information about the surrounding code and libraries: the types of the values used in the expression, the locals in scope, and the visible library methods and fields. The algorithm cannot merely return the entire list of completions because some partial expressions have an infinite list of completions. What constitutes a valid completion is defined by Figure 3.6.

The algorithm does completion finding and ranking simultaneously in order to compute the top n completions efficiently where the ranking function is the one defined in the previous section. To generate exactly n completions, the generator is called n times. In practice, the user interface would likely show 10 completions at first and allow the user to scroll down to examine more if desired.

²<http://www.vertigo.com/familyshow.aspx>

We present a general algorithm for computing ranked completions of a partial expression, first giving a naive implementation and then discussing optimizations. The main logic is Algorithm 3.1, which returns a generator that returns all completions of a partial expression in order by score. The **yield return** statement returns a single completion, and when the next completion is requested, execution continues on the next statement. The first n elements of `AllCompletions(\tilde{e})` are the top n ranked completions of \tilde{e} .

Note that for any partial expressions containing `.*`, this generator will usually continue producing more completions forever, but can be called only n times to get just the top n completions. It may be easier to first read the algorithm while ignoring the `score` variable, which is necessary to handle the unbounded result set for `.*`. Then it is a simple recursive algorithm which computes every possible completion of its subexpressions and uses those completions to generate every possible completion of the entire expression (e.g. all methods that can take those arguments). This simple structural recursion is driven directly by the input form of the partial expression.

Note that `?` is interpreted as `vars.*` where `vars` is a special subexpression whose list of completions is every local and global variable in scope.

We now discuss various useful optimizations.

3.4.1 Cache subexpression scores

A subexpression's score will be needed for every completion it appears in. To compute it only once, the algorithm is redefined such that it returns a set of pairs of completions and their scores.

3.4.2 Compute completions not in score order

In the algorithm given above, completions with a score not equal to `score` are discarded and regenerated later. To avoid that work, completions with a score greater than `score` can be saved to be output later. For most partial expressions, it makes sense to compute all of the completions for a given value of `concreteSubs` at once. In the case of `.*` queries,

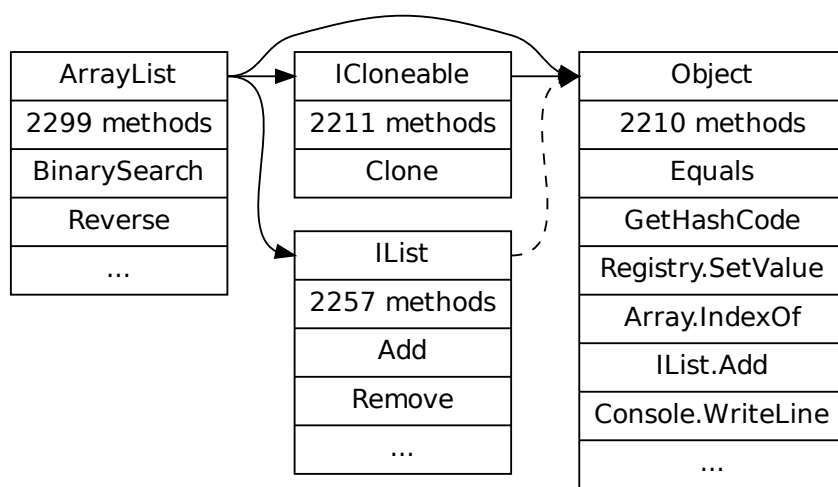


Figure 3.8: The method index. The supertypes of `IList` other than `Object` are omitted for simplicity.

the algorithm will never be done computing every possible completion, but `foo.*` can be thought of as the union of `.?` queries first on `foo`, then on the results of `foo.?`, etc. Then each completion set is finite and the basis for the future completions. In pseudocode, this is implemented as inserting each completion `c` into `subcomps`.

3.4.3 Indexing

As written, how the algorithm iterates over possible completions is unspecified. This is especially a problem for unknown methods as simply iterating over all methods in a huge framework would take too long. An index is maintained that maps every type to a set of methods for which at least one of the arguments may be of that type. Then, given a query like `?({ e_1, e_2 })`, each of the argument types is looked up to see how many methods would have to be considered for that type and the smallest set is chosen. That set will almost always be orders of magnitude smaller than the set of all methods.

Part of a method index is shown in Figure 3.8. In order to save memory, the method index is organized such that looking up a type τ gets a set of methods which have parameters of the exact type τ along with pointers to the method indexes for the immediate supertypes

of τ . Due to the type distance part of the ranking algorithm explained in Section 3.3, each method index visited will give progressively worse ranked results.

Methods are a prime candidate for indexing as there are many methods and few that take a specific type. Although the current implementation does not do so, queries for multiple field lookups could also be made more efficient using an index that indicates for each type which types are reachable by a `.*` query, how many lookups are needed, and which lookups can lead to a value of that type. For example, a `Line` type with `Point` fields `p1` and `p2` and a `GetLength()` method would have an entry denoting the closest `double` requires 2 lookups with the next lookup being one of `p1` or `p2` (followed by `.X` or `.Y`).

3.4.4 Avoid computing type-incorrect completions

If the possible valid types for the completions were known, then the type reachability index would be more useful: otherwise results of every type have to be generated anyway for completeness. At the top level, the context will often provide a type unless the expression being completed is the initial value for a variable annotated only as `var` (the C# syntax for local type inference) or is in a `void` or `Object` context, so nearly any type is valid. On the recursive step of the algorithm, the possible types may not be known or may not be precise enough to be useful: there are methods that take multiple arguments of type `Object`, so even knowing one of the argument types does not narrow down the possibilities for the rest. On the other hand, binary operators and assignments are relatively restrictive on which pairs of types are valid, so enumerating the types of the completions for one side could significantly narrow down the possibilities for the other side.

3.4.5 Grouping computations by type

Which completions are valid is determined solely by the types of the expressions involved. Hence, instead of considering every completion of every subexpression separately, the completions of each subexpression can be grouped by type after grouping by score to reduce the number of times the algorithm has to check if a given type is valid in a given position.

This also allows type distance computations to be done once for all subexpressions of the same types. Any remaining ranking features are computed separately for each completion as grouping by them is no faster than computing their terms of the ranking function.

3.5 Evaluation

We would like to demonstrate that partial expressions are easy for users to write and that they work well for expressing code completion queries. In light of the difficulty of gleaning meaning from a user study on an API discovery tool, we chose to not to perform a user study in favor of an experiment simulating running the tool at many places in actual code bases. As a result, we do not have experimental data on how easy it is to write a partial expression in an API discovery context; we can merely argue that we believe it is easy to use and note that the authors of CodeHint [12] chose to include partial expressions as part of their input as well. On the other hand, by running a very large number of simulated queries, we have fairly strong evidence that partial expressions are effective at selecting desired APIs.

3.5.1 Experiment setup

We implemented the partial expression completion algorithm using the Microsoft Research Common Compiler Infrastructure (CCI).³ CCI reads .NET binaries and decompiles them into a language resembling C#. Unfortunately, we were unable to work on actual source code because at the time the experiments were performed, no tools for analyzing the source code of C# programs existed—and even if they did exist, open source C# programs are relatively rare.

We performed experiments where our tool found expressions in mature software projects, removed some information to make those expressions into partial expressions, and ran our algorithm on those partial expressions to see where the real expression ranks in the results.

All experiments were run on a virtual machine allocated one core of a Core 2 Duo E8400

³<https://cciast.codeplex.com/>

3GHz processor and 1GB of RAM.

The source code for the experiment can be found at <https://pec.codeplex.com/>. That repository also includes a prototype UI implemented as a Visual Studio plugin, which unfortunately cannot be used anymore due to relying on deprecated Visual Studio APIs to perform the source code part of the analysis.

One complication is that abstract type inference involves precomputation across the entire codebase—including the expression being completed. To avoid this situation, we re-run abstract type inference for each expression, eliminating the expression and all code that follows it in the enclosing method; we do consider the rest of the program.

Outline We describe three case studies whose significance we have previously discussed in Section 3.1 and show that the ranking scheme is effective and the algorithm is efficient, thereby demonstrating that partial expressions are a good way to express code completion problems. After that, we analyze the importance of the individual ranking features in Section 3.5.5 to justify our choices in designing the ranking function.

3.5.2 Predicting Method Names

Our first experiment shows that queries consisting of one or two arguments can effectively find methods. We ran our analysis on 21,176 calls across parts of seven C# projects listed in Table 3.1. We generated queries by finding all calls with ≥ 2 arguments (including the receiver, if any) and giving one or two of the call’s arguments to the algorithm. We evaluated the algorithm on the rank it gave to the actual method. While putting the correct result as the first choice is ideal, we do not consider it necessary for usefulness since users can quickly skim several plausible results.

Figure 3.9 shows the results overall and partitioned between static and instance calls. Almost 85% of the time, the algorithm is able to give the correct method in the top 10 choices. An additional 5% of the time, the correct method appears in the next 10 choices out of a total of hundreds of choices on average.

Table 3.1: Summary of quality of best results for each call

Program	# calls	# top 10	# top 11..20
Paint.Net ^a	3188	2288	525
WiX ^b	13192	11430	512
GNOME Do ^c	208	167	22
Banshee ^d	91	82	2
.NET ^e	2801	2345	145
Family.Show ^f	586	510	23
LiveGeometry ^g	1110	1072	3
Totals	21176	17894 (84.5%)	1232 (5.8%)

^a <http://www.getpaint.net/>—image editor (main .exe)

^b <http://wix.codeplex.com/>—Windows Installer XML

^c <http://do.davebsd.com/>—application launcher

^d <http://banshee.fm/>—media player

^e .NET Framework v3.5 libraries System.Core.dll, mscorlib.dll

^f <http://www.vertigo.com/familyshow.aspx>—WPF example application

^g <http://livegeometry.codeplex.com/>—geometry visualizer

Notably, the algorithm fares significantly better on instance calls than static calls. This is not too surprising as the search space is much larger for static calls. This might also indicate that the current heuristics prefer instance calls more strongly than they should.

Unfortunately, we cannot algorithmically determine which argument subset a user would use as their search query. Instead, we show that usually for *some* set of no more than 2 arguments the correct method being highly ranked. Our intuition, which would need a user study to validate fully, is that evaluating our approach by choosing for the best possible subset of arguments is reasonable because programmers are capable of identifying the most useful arguments (e.g., `PreciseLibraryType` instead of `string` or `Pair`).

Figure 3.10 shows that a single argument is often enough for the algorithm to determine which method was desired, in this case defined as putting the method in the top 20 choices.

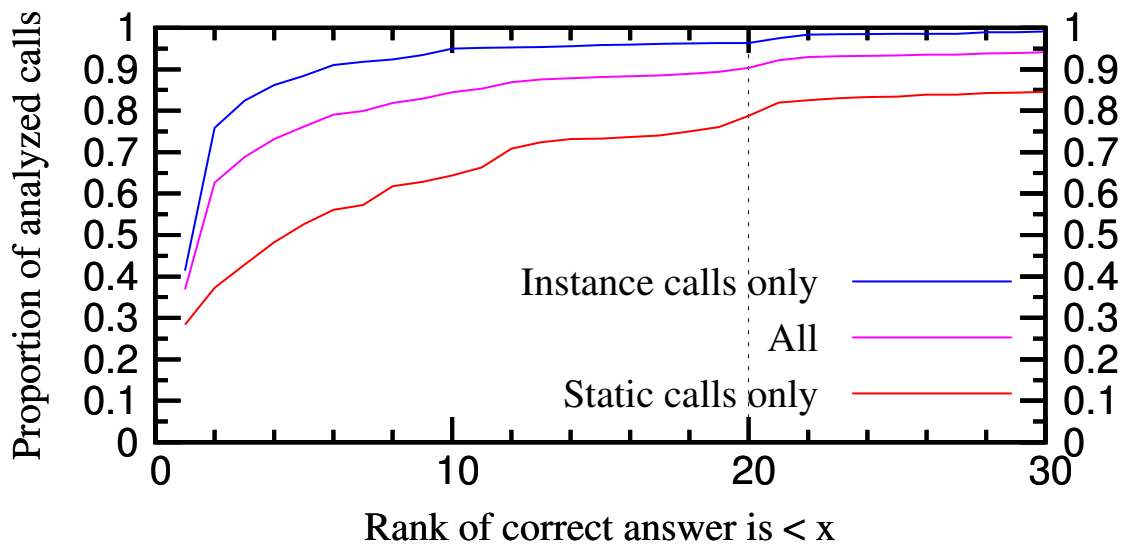


Figure 3.9: The proportion of calls of each type with the best rank of at least the given value

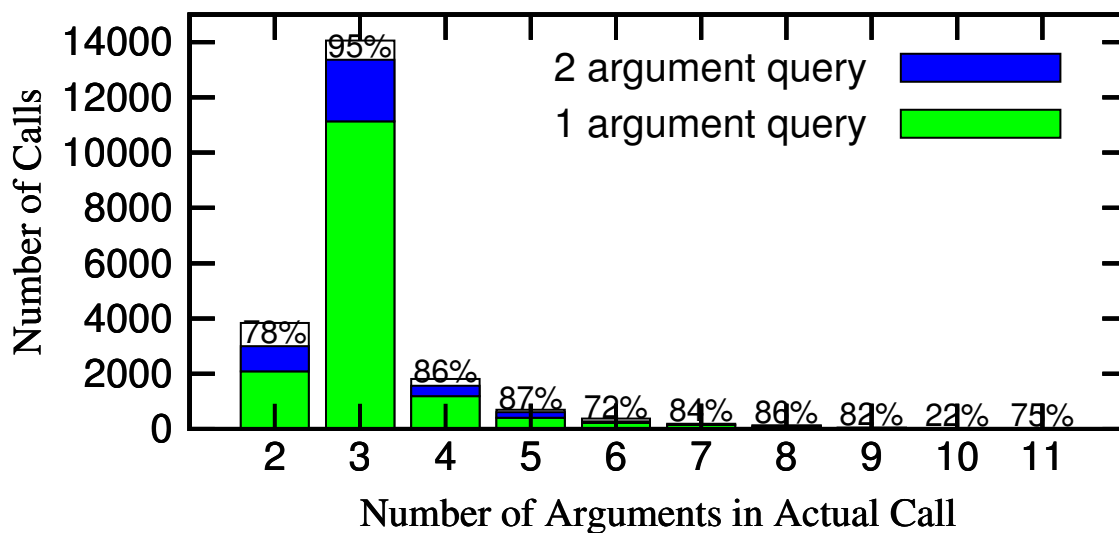


Figure 3.10: Each bar represents all calls analyzed with the number of arguments. The sections of the bar correspond to how many of those arguments the algorithm needed to put the original call in the top 20 results.

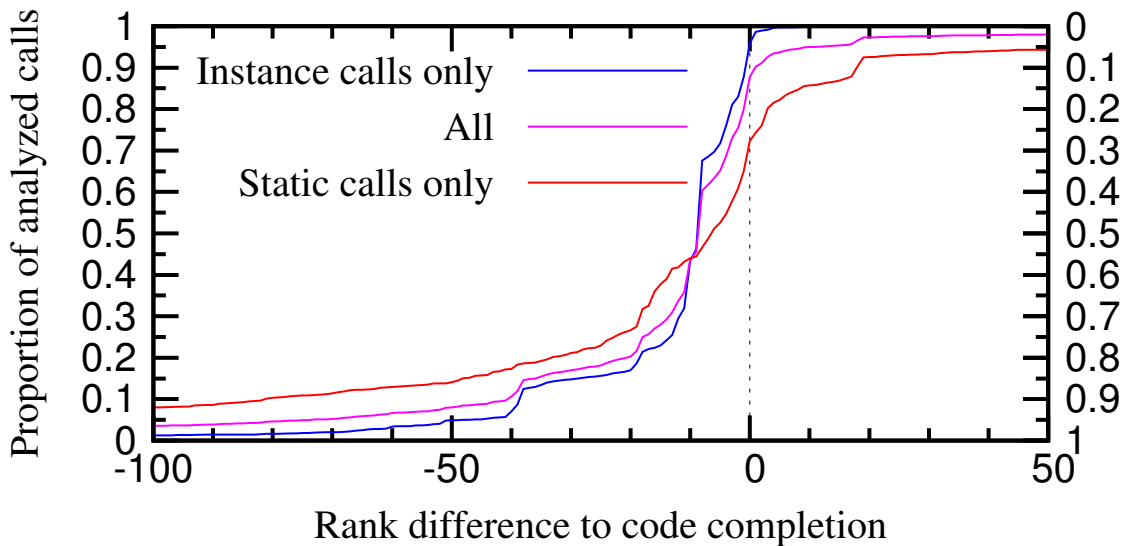


Figure 3.11: Difference in rank between our algorithm and Intellisense

Not shown in the graph is that adding a third argument leads to only negligible improvement in these results; note that even knowing all of the arguments to a method might not be enough to place it in the top 20 choices. Above the bars is the percentage of calls the algorithm was able to guess using only two arguments, which is high for any number of arguments. The intuition is that most of the arguments are not important, although there are also more opportunities for an argument to be of a rarely used type. The low value for 10 argument calls is due to there being very few such calls and most of them being from a large family of methods which all have the same method signature.

Comparison to code completion

In order to have a baseline, Figure 3.11 compares our ranking algorithm to Intellisense. The x -axis measures how much closer to the top of the ranking our algorithm put the answer than Intellisense's alphabetical ordering did where the extreme of -100 means using our system instead of Intellisense would have required reading through 100 fewer completions. The y -axes are read as the left side measuring the proportion of calls our system did better on

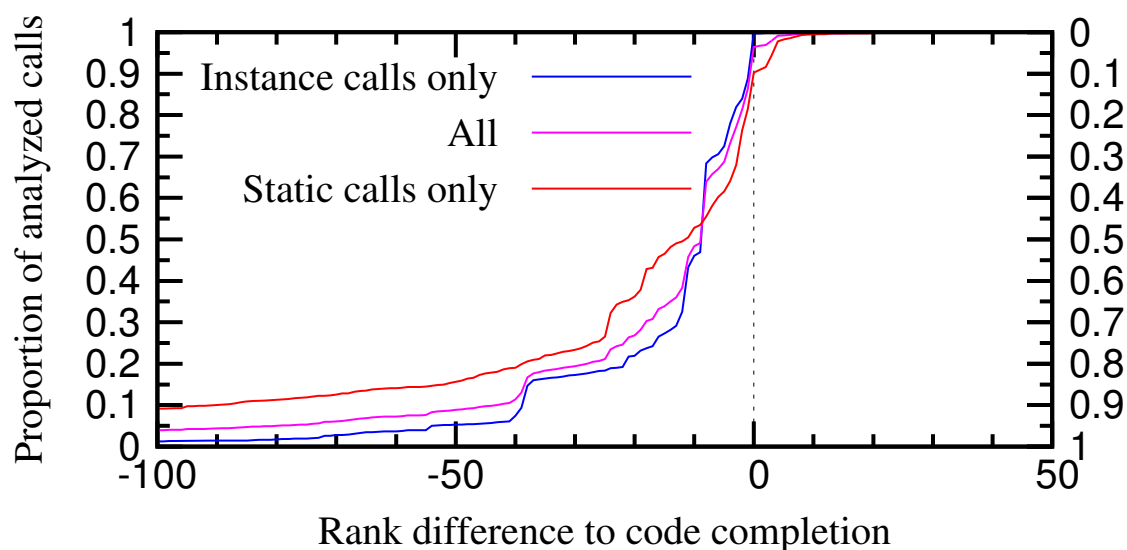


Figure 3.12: Difference in rank between our algorithm filtering its results for those matching the correct return type and Intellisense

and the right side measuring the proportion of calls Intellisense did better on. For example, at $x = -20$, the middle line is at 0.2 on the left y -axis, meaning that for 20% of the calls, our system placed the answer at least 20 ranks ahead of Intellisense, while reading the right y -axis tells us that for the other 80% of the calls, Intellisense did no worse than ranking the answer within 20 ranks of our system.

We modeled Intellisense as being given the receiver (or receiver type for static calls) and listing its members in alphabetic order. Intellisense knows which argument is the receiver but is not using knowledge of the arguments. It was considered to list only instance members for instance receivers and only static members for static receivers. Given this ordering, we were able to compute the rank in the alphabetic list of the correct answer. We then subtracted that rank from the rank given by our algorithm, so negative numbers mean our algorithm gave the correct answer a higher rank.

About 45% of the time, our position is at least 10 higher than it is with Intellisense. Since Intellisense displays at most 10 results at a time, this means it is not initially displayed by Intellisense.

Subtracting the ranks is oversimplifying the comparison. First, the different tools have different inputs. Our tool does not require the receiver but is helped by being provided a second argument. Second, the Intellisense results are listed in alphabetic order which is likely easier to skim through than the results from our tool which will be ordered by their ranking scores. Last, the improvement of going from 11th to 1st seems a lot more valuable than going from 111th to 101st, although both are improvements by 10 ranks. The take-away is not that our tool is “better” than Intellisense; they serve different purposes. Instead, we wanted to show that our tool is often able to greatly reduce the number of choices a user would have to sift through compared to Intellisense even if the user knew the correct receiver.

Filtering on return type Figure 3.12 shows a similar comparison to the one in Figure 3.11 except that our algorithm additionally knew the desired return type (or `void`) and only suggested methods whose return type matched.

The assumption of a known return type is not used elsewhere both because, in the context of API discovery, the user may often not know what return type to use, and the `var` keyword in C# and equivalents in other languages allow a user to omit return types. Furthermore, the assumption is used in related work on API discovery, so we wanted to show partial expressions could discover completions effectively without even needing to know the type, although they could be aided by additionally knowing the type.

Speed For 98.9% of the calls analyzed, the query with the best result ran in under half a second, which is fast enough for interactive use.

As a caveat, these times do not include running the abstract type inference algorithm. That could take as long as several minutes for a large codebase but can be done incrementally in the background.

These times were measured using CCI reading binaries as opposed to getting the information from an IDE’s incremental compiler, but we observed similar performance from our prototype IDE plugin. Additionally, any such effects were minimized by memoizing a lot of

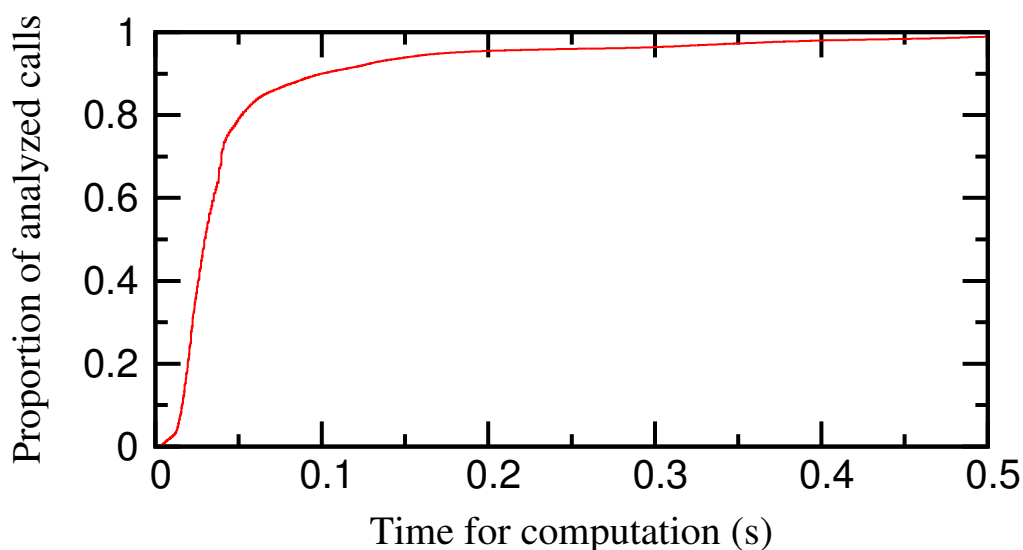


Figure 3.13: Computation time for the run of the algorithm with the best result for each call

the information from CCI, so the vast majority of the time was spent in our algorithm.

3.5.3 Predicting Method Arguments

Our second experiment investigated how often arguments to a method could be filled in by knowing their type. Due to the design of our ranking function, knowing what method the expression was an argument to also may give a useful abstract type.

Looking at the same method calls as the previous experiment, for each argument in each call, a query was generated with that argument replaced with `?`. There were a total of 69,927 arguments across the 21,176 calls. 23,927 were considered not guessable due to having an expression form that our partial expression completer does not generate like an array lookup or a constant value.

Figure 3.14 shows how well our algorithm is able to predict method arguments, with the lower line ignoring the low-hanging fruit of local variables. Over 80% of the time, the algorithm is able to suggest the intended argument as one of the top 10 choices given out of an average of hundreds of choices.

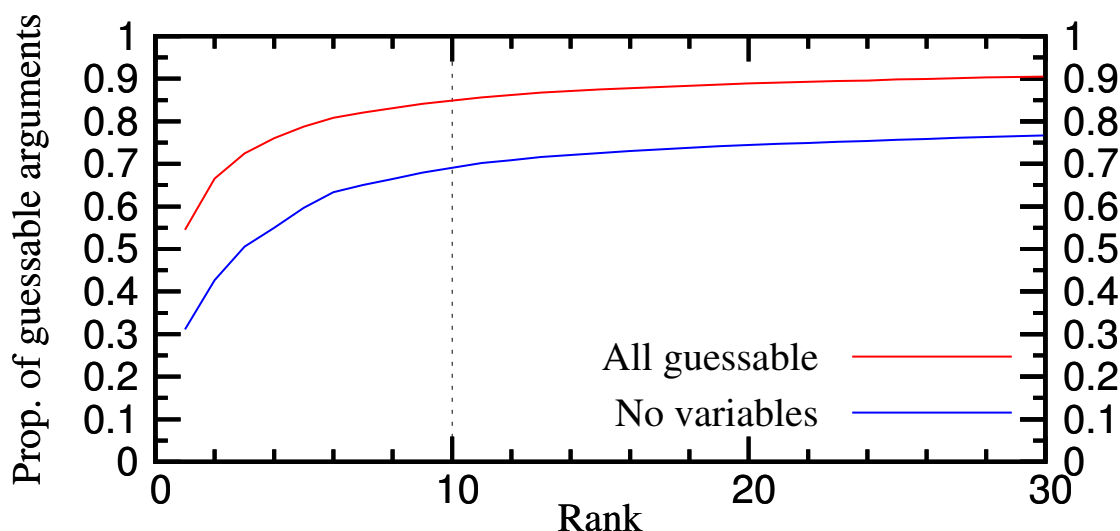


Figure 3.14: The proportion of guessable arguments which could be guessed with a given rank and the same ignoring arguments which are just local variable references

Use of expressions other than local variables in argument positions is common as shown in Figure 3.15. Programmers must somehow discover the proper APIs for these expressions: Intellisense only suggests local variables given an argument position. The “not guessable” expressions are those that involve constants or computation like an addition or a non-zero argument method call that could be guessed by neither our technique nor Intellisense. Our partial expression language captures more of the expressions programmers use as arguments including field/property lookups which are relatively common and require browsing to find using Intellisense.

As our experiments are on decompiled binaries and not the original source, these arguments may not be exactly what the programmer wrote. In particular, expressions might be stored in temporary variables that they did not write or temporaries they did write might be removed, putting their definition in an argument position. By inspection, it appears this is not the case for the code we worked on.

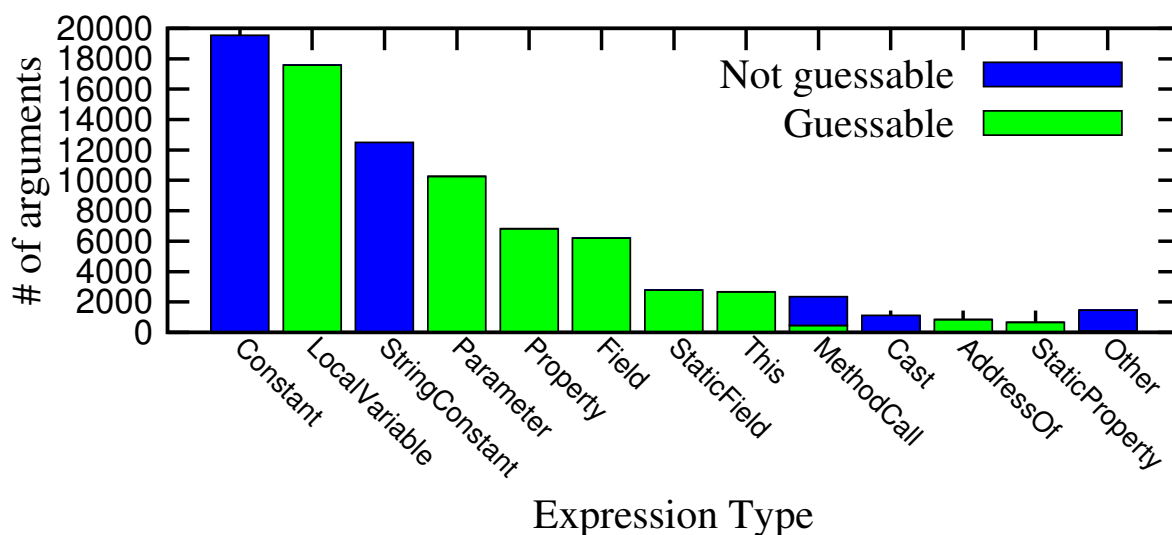


Figure 3.15: The kinds of expressions found in argument positions, showing which ones are considered guessable

Speed Our tool is capable of enumerating suggested arguments in under a tenth of a second 92% of the time and under half a second over 98% of the time, which is fast enough for an interactive tool.

3.5.4 Predicting Field Lookups

Our third experiment determines how often field/property lookups could be omitted in assignments and comparisons (on either side).

We found all assignments and comparisons with a field/property lookup on either side, removed one or both of the lookups and put `.?` or `?.?` respectively at the end of both sides. Then the rank of the original expression in the results was recorded. Only one `.?` was used in the assignment part of the experiment for efficiency reasons; comparisons are valid for only a few types, so many fewer possible results had to be considered. Note that our implementation generates and scores all results in order to ensure that it finds the original expression as opposed to stopping after finding the top n which would be significantly faster for small n . Additionally, due to the structuring of the ranking function, results with two

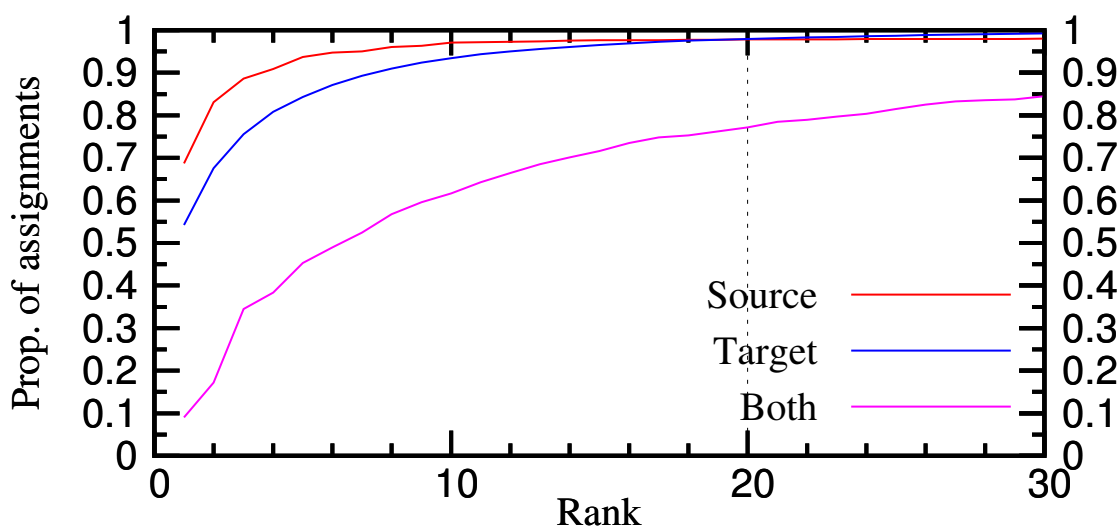


Figure 3.16: Proportion of assignments where a field lookup could be removed from one or both sides and guessed with a given rank

lookups will usually show up after results with one lookup anyway.

Our corpus includes 14,004 assignments where the target ends with a field lookup, 7,074 where the source does, and 966 where both do. For those assignments, Figure 3.16 shows the rank of the correct answer when our algorithm was given the assignment with the final field lookups removed and $\underline{.?}$ added to the end of both sides of the assignment. The correct answer was in the top 10 choices over 90% of the time when one field lookup was removed, but only about 59% of the time when a field lookup was removed from both sides, going up to 75% when considering the top 20 choices. There were a total of dozens of choices on average.

Our corpus includes 620 comparisons where the left side ends in a lookup, 162 of which end in two lookups; 620 comparisons where the right side ends in a lookup, 174 of which end in two lookups; and 125 where both sides end in a lookup. Of those, Figure 3.17 shows the ranks our tool gave to the expression in the source given the query containing the original expression with the lookups removed and $\underline{.?.?}$ added to the end of both sides.

The numbers are significantly better than for assignments because there are fewer pos-

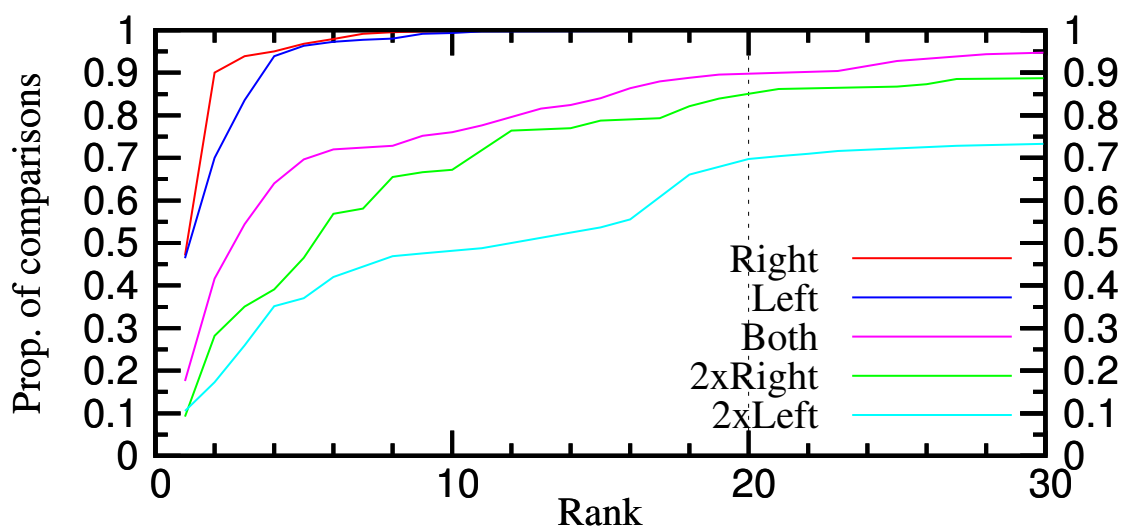


Figure 3.17: Proportion of comparisons where field lookups could be removed from one or both sides and guessed with a given rank

sibilities: few types support comparisons. One lookup can be placed within the top 10 for nearly every instance in our corpus. If we allow 20 choices, then two lookups where one lookup is on each side can be guessed 89% of the time while two lookups on the same side can be guessed 85% of the time if they are on right and 69% of the time if they are on the left. The discrepancy between the latter two appears to be that comparisons against constants are usually written with the complicated expression on the left and the constant on the right, and the name matching feature is not helpful for comparisons to constants.

Speed 99.5% of these queries ran in under half a second.

3.5.5 Justifying the ranking function

We ran two experiments to determine if the ranking function we chose was actually the best choice. First we tried to modify it by using only a subset of the terms. Second we tried to generate a brand new ranking function using machine learning. Our results show that the pieces we expect to be valuable actually are and that our ranking function cannot be

easily improved upon. Notably, other projects do collect additional information by looking at existing code; our abstract type inference does so, but we do not collect any more in-depth information than that which could possibly improve the ranking function.

Sensitivity analysis of ranking function

To see which parts of the ranking function were most important, we re-ran the experiments with various modified ranking functions. Each modified version either included only one of the terms or left out one of the terms, in addition to versions that left out and included both the type distance term and the abstract type term. Table 3.2 shows the data for the proportion of expressions where the correct answer was in the top 20 choices for different variants of the ranking function for each of the experiments.

Methods Type distance and abstract type distance are the only features that matter. Leaving out the namespace and in-scope static terms seems to make almost no difference. Furthermore, the two type distance terms separately are both good, with abstract type distance alone being a little better, but not quite as good as the two together, confirming that both are useful.

Arguments It seems that only the depth feature seems to matter. Leaving it out makes the results much worse while leaving any other term out has almost no effect. In fact, looking at the “+d” column, using just the depth term gives almost exactly the same results as the full ranking scheme.

Assignments For just one lookup removed from either side, once again only depth matters, but when a lookup is missing from both sides, the type distance computation becomes important. In fact, in the case of a lookup missing from both sides, leaving out the depth component improves the results. This is not too surprising as there are likely many possible assignments which require adding only one lookup to either side. The interesting part is

	Count	All	-n	-s	-d	-m	-t	-a	-at	+n	+s	+d	+m	+t	+a	+at
Methods																
All	21176	0.90	0.90	0.90	-	-	0.85	0.84	0.43	0.43	0.43	-	-	0.84	0.85	0.90
Instance	13904	0.96	0.96	0.96	-	-	0.87	0.94	0.31	0.31	0.31	-	-	0.94	0.87	0.96
Static	7272	0.78	0.78	0.78	-	-	0.80	0.65	0.65	0.68	0.65	-	-	0.64	0.81	0.78
Arguments																
Normal	45325	0.90	0.90	-	0.72	-	0.90	0.90	0.90	0.72	-	0.90	-	0.72	0.72	0.72
No variables	14925	0.77	0.77	-	0.65	-	0.76	0.77	0.76	0.64	-	0.76	-	0.65	0.64	0.65
Assignments																
Target	14004	0.97	-	-	0.87	-	0.97	0.97	0.97	-	-	0.97	-	0.81	0.87	0.87
Source	7074	0.89	-	-	0.87	-	0.90	0.89	0.90	-	-	0.90	-	0.87	0.89	0.87
Both	966	0.75	-	-	0.76	-	0.74	0.75	0.73	-	-	0.73	-	0.75	0.75	0.76
Comparisons																
Left	620	1.00	-	-	0.23	1.00	1.00	1.00	1.00	-	-	1.00	0.65	0.25	0.68	0.25
Right	620	1.00	-	-	0.51	1.00	1.00	1.00	1.00	-	-	1.00	0.53	0.62	0.85	0.62
Both	125	0.89	-	-	0.46	0.87	0.88	0.89	0.88	-	-	0.87	0.46	0.42	0.37	0.42
2xLeft	162	0.69	-	-	0.14	0.69	0.70	0.69	0.70	-	-	0.69	0.41	0.18	0.57	0.18
2xRight	174	0.85	-	-	0.63	0.81	0.81	0.85	0.81	-	-	0.77	0.58	0.71	0.73	0.71

Table 3.2: Ranking function term sensitivity. Each cell is the proportion where correct answer was found in the top 20 choices with various modifications of the ranking function. “All” is the full ranking function. For the rest, - means without certain terms, + means with only certain terms: ‘n’=namespaces, ‘s’=s=in-scope static, ‘d’=depth, ‘m’=matching name, ‘t’=t=normal type distance, and ‘a’=abstract type distance.

that apparently these lookups can be distinguished from the proper one by looking at more detailed type information (recall that only assignments which are type correct are even being considered).

Comparisons Depth once again seems to be most important. Except on the “2xRight” row, depth appears to be the only significant feature. The different values for that row vary little, indicating that each ranking feature is somewhat useful, but there is little gain from combining them. On average, there were hundreds of type-correct options, so the ranking function is definitely doing something to place the correct option in the top 20.

Learning a better ranking function

Simply leaving out parts of the ranking function helps show which parts are useful, but is not enough to show that the ranking function is actually a good one. For instance, maybe we have the right pieces, but the coefficients are wrong. In order to verify that we actually did choose a good ranking function, we used machine learning to attempt to learn a better ranking function.

A key piece of using machine learning is the proper selection of features. We brainstormed a list of features (Table 3.3) for the ranking function to consider, but a better ranking function may exist that uses features we did not think of or did not collect data for. Specifically, none of our features involve learning from other code to, for example, see what APIs are commonly combined.

Setup Multiple machine learning techniques were used to learn a new ranking function. Gradient boosted regression trees were learned using pGBRT⁴ and SVMs were learned using SVM^{rank5} with both linear and radial basis kernels. Various parameters were tried for the different methods.

⁴<http://machinelearning.wustl.edu/index.php/main/pGBRT>

⁵http://www.cs.cornell.edu/People/tj/svm_light/svm_rank.html

#	Context	Description
1	Result	type distance
2	Result	empty prefix (number of missing arguments before first filled in argument)
3	Result	empty (total number of missing arguments)
4	Result	empty value type (number of missing arguments of value (non-reference) types)
5	Result	method full name (including namespace and type name) length in characters
6	Result	method full name (including namespace and type name) # of uppercase characters
7	Result	method name length in characters
8	Result	method name # of uppercase characters
9	Result	method parameter count
10	Result	method is static
11	Result	method is constructor
12	Containing type	accessibility (enum value from Roslyn)
13	Containing type	accessibility is public
14	Containing type	is static
15	Containing type	is an inner type
16	Containing type	number of dots (".") in namespace
17	Containing type	type distance to object
18	Containing method	accessibility (enum value from Roslyn)
19	Containing method	accessibility is public
20	Containing method	is static
21	Containing method	is an override
22	Containing method	number of parameters
23	Containing method	arity (number of type parameters)
24	Containing method	is a constructor
25	Containing method	is a static constructor
26	Containing method	is a property getter
27	Containing method	is a property setter
28	Containing method	is an ordinary method
29	Containing method	returns void
30	Containing method	return type's type distance to object (0 if void)
31	Query	number of locals declared before query
32	Query	expression depth (steps up in parse tree to closest non-expression)
33	Query	nesting depth (number of blocks contained in which are not the method block)
34	Query	line number within method
35	Query	query in an expression statement
36	Query	query in an argument
37	Query	query on right side of a local declared using the <code>var</code> keyword
38	Query	query on right side of a local declaration not using the <code>var</code> keyword
39	Query	query on right side of an assignment
40	Query	query is an argument to an arithmetic operation
41	Query	query is an argument of a logical operation
42	Query	query is an if condition
43	Query	query is a while condition
44	Query	query in unknown context
45	Query	query on right side of a field declaration
46	Query	query is the expression in a lambda
47	Query	query is the expression of a <code>RETURN</code>
48	Query	query is the receiver of a method/field/property lookup
49	Query	query is used in a <code>foreach</code> loop
50	Query	query is in an initializer block

Table 3.3: Features for learned ranking function

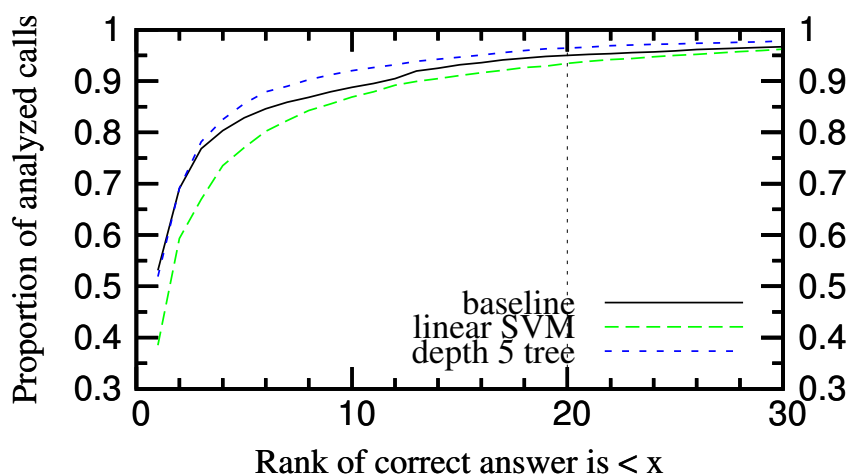


Figure 3.18: Performance of learned ranking functions on the test set

As learning with the SVM using the radial basis function and polynomial kernels was very slow (ran for days without finishing on a smaller version of the dataset), it was trained on a randomly selected subset of one hundredth of the training data used for the other methods.

Results Figure 3.18 compares the best SVM and best regression tree performance on the test set and shows that overall the SVM does slightly worse and the regression trees do slightly better than the baseline. While it is a small increase, the regression tree does enough better than our baseline to be interesting.

Figure 3.19 shows a direct by-query comparison of the regression tree to the baseline algorithm. Points below the line mean that the regression tree did better while points above the line mean that the baseline algorithm did better.

Analysis One good thing about regression trees is that they are readable. The tree learned can be seen in Figure 3.20 and could be useful for informing a better ranking function.

Due to the implementation of the query engine, actually plugging in an arbitrary ranking function is non-trivial. Specifically, to get reasonable performance, it is important to be able to output results before scoring every possible result. Also, a simpler algorithm might

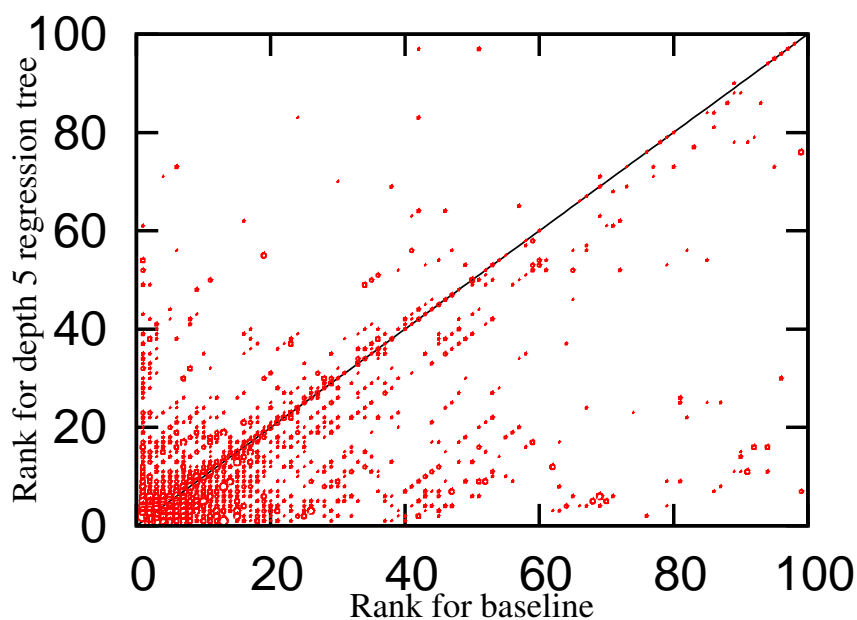


Figure 3.19: By query comparison of the rank of the baseline algorithm versus the regression tree. The line is parity.

be better due to being more likely to be understandable to the user, although the current algorithm may already be too complicated for that.

Looking at the regression tree in Figure 3.20, some interesting facts are revealed that may be helpful in generating an improved ranking algorithm:

1. Type distance is only ever compared to 1 (the minimum is 0, so < 1 is the same as $= 0$). Perhaps including type distance as a linear term in ranking function is overly strongly penalizing methods which are not exact type matches.
2. The method name length features got used a lot. This seems likely to be overfitting, but the algorithm works, so perhaps longer method names correspond to certain types of methods.
3. The first split is on whether the result is a static method. An earlier iteration of the ranking algorithm used whether the result was static or not as looking at results

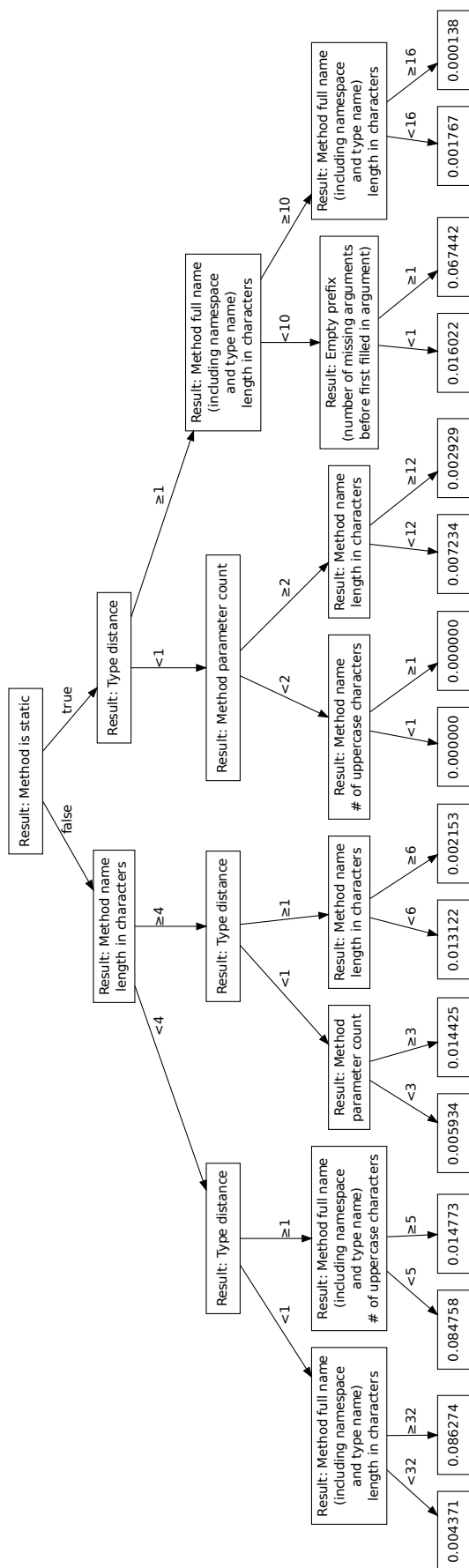


Figure 3.20: The learned decision tree ranking function

it seemed necessary, but analysis showed that feature was not improving the results. This implies that it is important to consider that information in the ranking.

4. All of the features in the tree are result features (as are all of the features in the current algorithm). This gives evidence that trying to use the context of the query is not very useful for ranking the results.

3.6 Conclusion

To address the problem of API discovery, we developed the concept of partial expressions, a superset of the base programming language, allowing the programmer to state queries capturing as much of their knowledge as possible in a familiar syntax. We showed these queries are effective at completing expressions across multiple C# libraries. We also verified that the ranking function we developed is well chosen.

Chapter 4

TEST-DRIVEN SYNTHESIS

Our Test-Driven Synthesis concept explores how much a programming-by-example system can get out of treating the order of the examples as part of the description of the problem. The intuition is that a user describing a problem with examples will naturally provide examples of simple behavior before more complicated behavior. We show an algorithm that is able to take advantage of the order by building a program for the simple behavior and then modifying it to encompass the more complicated behavior as well. By using this small amount of additional input that the user is able to provide easily, we were able to develop a relatively straightforward program synthesis algorithm that works across multiple domains.

We call our system **LaSy**¹. Taking inspiration from the test-driven development (TDD) [22, 36] concept of iterating on a program that always works for the tests written so far, the behavior of functions in **LaSy** is constrained by a *sequence* of increasingly sophisticated examples like those that might be written by a programmer authoring a function using the TDD programming methodology or demonstrating the functionality via examples to another human.

Prior PBE systems take as input a *set* of *representative* examples from which a program is synthesized (they may be consumed one at a time, but their order is not prioritized). It is desirable that the user provides representative examples; otherwise the system may end up over-fitting on a simple example, and would fail to find a common program that works for all examples [28]. On the other hand, when describing a task we conjecture people can easily provide a *sequence* of examples of increasing complexity—for example, in a human-readable explanation [44] or in a machine-readable sequence of tests for TDD. The

¹Language for Synthesis, pronounced “lazy”

presentation as a sequence allows for the problem of learning the task to be broken down into a series of subproblems of learning slightly more detail about the task after seeing each new example—this is one of the key insights of this work. While this bears some similarity to genetic programming [38], our system does not rely on any concept of a function almost or approximately matching an example. By working in a series of steps where after each step the result is a program that is correct for a subset of the input space, our system is able to more effectively narrow down the search space than prior work.

4.1 *Motivating examples*

We present examples demonstrating the breadth of programs LaSy is able to synthesize to highlight its domain-agnostic capabilities. Section 4.5 goes into more detail on evaluating the performance and design decisions of our synthesizer.

4.1.1 *Automating TDD*

To demonstrate iteratively building up a program, we use the example of greedy word wrap, whose use as a TDD practice problem [36] was the original inspiration for our methodology. Word wrap inserts newlines into a string so that no line exceeds a given maximum line length; the greedy version inserts newlines as late as possible instead of trying to even out the line lengths.

Figure 4.1 shows a LaSy program for implementing greedy word wrap. The LaSy program begins with a declaration saying to use the “`strings`” DSL followed by the function signature for word wrap. Most of the program is the list of examples that dictate how the function should behave.

The examples are interspersed with comments that explain how this sequence iteratively builds up a solution with intermediate steps where it is easy to describe what subset of the inputs the program is correct for. Exactly where to break the line is refined over several examples: first it is just at the space, then at the space in a string longer than the maximum line length, then at the last space in a long string, then at the last space before (or at) the

```

language strings;
function string WordWrap(string text, int length);

// Single word doesn't wrap.
require WordWrap("Word", 4) == "Word";
require WordWrap("Word", 100) == "Word";
require WordWrap("Word", 5) == "Word";
require WordWrap("WorW", 4) == "WorW";
// Two words wrap...
require WordWrap("Extremely longWords", 14) == "Extremely\nlongWords";
require WordWrap("How are", 4) == "How\nare";
require WordWrap("How are", 3) == "How\nare";
// ...when longer than line.
require WordWrap("How are", 76) == "How are";
require WordWrap("How are", 7) == "How are";
// Wrap as late as possible...
require WordWrap("How are you?", 9) == "How are\nyou?";
require WordWrap("How are you?", 8) == "How are\nyou?";
// ...but no later.
require WordWrap("Hello, how are youLongWord and more", 21) == "Hello, how are\n
youLongWord and more";
require WordWrap("Hello, how are you today?", 16) == "Hello, how are\nyou today?"
require WordWrap("Hello, how are you today?", 15) == "Hello, how are\nyou today?"
require WordWrap("Hello, how are you today?", 14) == "Hello, how are\nyou today?"
require WordWrap("How are you?", 7) == "How are\nyou?";
require WordWrap("Extremely longW ords", 14) == "Extremely\nlongW ords";
// Wrap in middle of word.
require WordWrap("abcdef", 3) == "Abc\ndef";
require WordWrap("abcd", 2) == "Ab\ncd";
require WordWrap("abcdef", 5) == "Abcde\nf";
require WordWrap("ThisIsAVeryLongWord a", 15) == "ThisIsAVeryLong\nWord a";
// Wrap multiple times (using recursion).
require WordWrap("ThisIsAVeryLongWord", 4) == "This\nIsAV\neryL\nongW\nord";
require WordWrap("abcdef", 2) == "Ab\ncd\nef";
require WordWrap("How are you?", 4) == "How\nare\nyou?";
// Complicated test to ensure program is correct.
require WordWrap("This is a longer test sentence. a bc", 7) == "This is\na\n
longer\ntest\nsentenc\n. a bc";

```

Figure 4.1: LaSy program for greedy word wrap

maximum line length, only to be refined further toward the end to include inserting breaks in the middle of words that do not fit on a single line. Also, our synthesizer is able to generalize from inserting a single line break to inserting any number of line breaks via recursion.

The high number of examples indicates this is a fairly sophisticated program for PBE; it should be noted that full test coverage for TDD takes a similar number of examples [36], and, furthermore, that this was the first example sequence we came up with that worked, not the shortest one. We include word wrap to show LaSy is capable of scaling up to that many examples.

The actual usefulness of the algorithm for this application is unclear: a programmer could just write the code, probably with less effort than writing the test cases. One could imagine using the system to force a programmer to actually provide good test coverage, although this probably makes more sense as a teaching tool than in an actual software development environment. One could imagine a software development class giving an assignment to teach test writing in which a student is given a specification and instead of being required to write the code, is required to write a test case sequence that leads our synthesizer to write the code.

4.1.2 End-user data transformations

LaSy can be used to synthesize useful scripts in many different domains for end-user data transformations; we show examples of two:

String transformations Consider the task of converting bibliography entries (represented as strings) from one format to another. Figure 4.2 shows an example of a LaSy program for converting between two such formats. This program is structured using helper functions for rewriting the authors list and for mapping venue abbreviations to the full venue names used in the target format. Unlike `ConvertName` and `ConvertList`, `VenueFullName` does not do any computation (the full name for "POPL 2013" cannot be inferred from the full name for "PLDI 2012" without a web search), so it is declared as a lookup, meaning the function will

```

language strings;

function string ConvertBib(string oldFormat);
function string ConvertName(string fullName);
function string ConvertList(string oldFormat);
lookup string VenueFullName(string abbr);

require ConvertName("John Smith") == "Smith, J.";
require ConvertName("Ann Miller") == "Miller, A.";

require ConvertList("John Smith") == "Smith, J.";
require ConvertList("Donna Jones, John Smith, Ann Miller") == "Jones, D.; Smith,
J.; Miller, A.";

require VenueFullName("PLDI 2012") == "The 33rd ACM SIGPLAN conference on
Programming Language Design and Implementation, Beijing, June, 2012";
require ConvertBib("Reagents: Expressing and Composing Fine-grained Concurrency,
PLDI 2012, Aaron Turon") == "Reagents: Expressing and Composing Fine-grained
Concurrency\nTuron, A.\nThe 33rd ACM SIGPLAN conference on Programming Language
Design and Implementation, Beijing, June, 2012";

require VenueFullName("CACM 2012") == "Communications of the ACM, 2012";
require ConvertBib("Spreadsheet Data Manipulation using Examples, CACM 2012,
Sumit Gulwani, William Harris, Rishabh Singh") == "Spreadsheet Data Manipulation
using Examples\nGulwani, S.; Harris, W.; Singh, R.\nCommunications of the ACM,
2012";

```

Figure 4.2: LaSy program for converting bibliography entries

just store the list of input/output examples and look up any inputs in that list to find the corresponding output.

XML transformations We show two LaSy programs for XML transformation tasks found on help forums. Figure 4.3 takes a set of `<div>`s containing named paragraphs and arranges the data as a table with a column for each `<div>` and a row for each name, so data is lined up in a row if the same name appears in multiple `<div>`s. This transformation requires a helper function which describes how a table row is built from a paragraph name. Both functions are simple enough that they require only a single example. Figure 4.4 assigns class attributes to paragraphs without them according to the class of the nearest previous paragraph (if present), and is implemented by the synthesizer using a loop.

4.2 Language

In addition to the LaSy programming-by-example language demonstrated in the previous section, which is explained in more detail in Section 4.2.1, we also discuss the language that experts use to define DSLs for use in LaSy in Section 4.2.2.

4.2.1 LaSy programming-by-example language

Figure 4.5 gives the syntax of LaSy. Note that LaSy relies on a base language, C# in our implementation, to provide basic types, functions, and values, and therefore the precise syntax for values and identifiers is omitted: type references are C# type references and values are C# expressions.

Programs in LaSy consist of a set of function declarations and a sequence of examples.

Language All LaSy programs begin with a reference to the DSL being synthesized over. The language is defined beforehand by an expert as described below in Section 4.2.2.

```

oldXml = "<doc><div id="ch1"> <p name="a1">1st Alinea.</p> <p name="a1.1">Zomaar
ertussen.</p> <p name="a2">2nd Alinea.</p> <p name="a3">3rd Alinea.</p> </div>
<div id="ch2"> <p name="a1">First Para.</p> <p name="a2">Second Para.</p> <p
name="a2.1">Something added here.</p> <p name="a3">Third Para.</p> </div></doc>";

language xml;

function XDocument ToTable(XDocument oldXml);
function XElement BuildRow(XDocument oldXml,
    string rowName);

require BuildRow(oldXml, "a1.1") == "<tr><td>Zomaar ertussen.</td><td/></tr>";
require ToTable(oldXml) == "<table>
<tr><td>1st Alinea.</td><td>First Para.</td></tr> <tr><td>Zomaar
ertussen.</td><td/></tr> <tr><td>2nd Alinea.</td><td>Second Para.</td></tr>
<tr><td/><td>Something added here.</td></tr> <tr><td>3rd Alinea.</td><td>Third
Para.</td></tr> </table>";

```

Figure 4.3: LaSy program for converting set of XML lists to a table

```

language XML;

function XDocument AddClasses(XDocument oldXml);

require AddClasses("<doc> <p>1</p> </doc>") == "<doc> <p>1</p> </doc>";
require AddClasses("<doc> <p>1</p> <p class='a'>2</p> <p>3</p> <p>4</p> <p
class='b'>5</p> <p>6</p> <p class='c'>7</p> </doc>")
== "<doc> <p>1</p> <p class='a'>2</p> <p class='a'>3</p> <p class='a'>4</p> <p
class='b'>5</p> <p class='b'>6</p> <p class='c'>7</p> </doc>";

```

Figure 4.4: LaSy program for adding class attributes

P	::= language I ; F * E *	(<u>P</u> rogram)
F	::= function t f ((t x ,) *);	(<u>F</u> unction declaration)
	lookup t f ((t x ,) *);	(<u>L</u> ookup declaration)
E	::= require f ((V ,) *) == V ;	(<u>E</u> xample)
V	::= any constant expression	(<u>V</u> alue)
t	::= I I <(t ,) * >	(<u>t</u> ype name)
f	::= I	(<u>f</u> unction name)
x	::= I	(<u>v</u> ariable name)
I	::= any valid identifier	(<u>I</u> dentifer)

Figure 4.5: The LaSy language.

Functions The function declarations list the functions to be synthesized. Each function has a name and type signature.

Examples and semantics Each example is a function call with literals given for its arguments and its required return value. A function f is considered to satisfy an example $f(V_1, \dots) == V_R$ if whenever f is called with arguments that are structurally equivalent (`.Equals()` in C#) to V_1, \dots , it returns a value that is structurally equivalent to V_R .

Specifically, the examples are an ordered list dictating a sequence of program synthesis operations wherein the function the example references is modified to satisfy the example without violating any previous example. At the beginning of the synthesis of a LaSy program, all functions are empty (and therefore satisfy no examples). The synthesis process is described in detail in Section 4.3.

The semantics of LaSy are naturally quite weak: the only guarantee is that if the synthesis succeeds, then the examples will be satisfied. The synthesizer is heavily biased toward smaller programs with fewer conditionals which tend to be more generalizable, but does not guarantee it will find the smallest program satisfying all of the examples. While the synthesizer implementation should act in a manner predictable enough that the user can trust its programs to be reasonable, ultimately only the user can determine if the synthesized program actually fulfills the user's intended purpose.

The result of the synthesizer is C# code, which can be compiled and used in any .NET program, including another LaSy program.

4.2.2 DSL definition language

An example DSL definition is given in Figure 4.6. The DSL gives a grammar that specifies what programs are possible as well as what the semantics of those programs are. Additionally, the DSL optionally provides a few different kinds of hints to the synthesizer that allow the synthesizer to take advantage of expert knowledge of the semantics.


```

language strings;
assembly flashfill.dll class lasy.FlashFill;
start P;
P ::= _CONDITIONAL(b, e);
b ::= |(d, ..., d);
d ::= &&( $\pi$ , ...,  $\pi$ );
 $\pi$  ::= m | !(m);
m ::= Match(v, r, k) | Match(f, r, k) | <(i, i);
i ::= Length(v) | GetPosition(v, p) | j;
j ::= _PARAM;
e ::= Concatenate(f, ..., f)
    | SplitAndMerge(v, s, s,  $\lambda$ f:e);
f ::= ConstStr(s) | SubStr(v,p,p) | Loop( $\lambda$ w:e)
    | SubStr(f,p,p) | Trim(f)
    | _LASY_FN(f) | _RECURSE(f, j);
s ::= _CONSTANT;
p ::= Pos(r,r,c) | CPos(k)
    | CPos(c) | CPos(j) | RelPos(p,r,c);
c ::= k | k*w+k;
k ::= _CONSTANT;
r ::= TokenSeq(T,...,T) |  $\epsilon$ ;
v ::= _PARAM;

rewrite &&( $\pi_0$ ,  $\pi_1$ ) ==> &&( $\pi_1$ ,  $\pi_0$ );
rewrite |(d_0, d_0) ==> d_0;
rewrite 0*w_0+k_0 ==> k_0;
rewrite Trim(Trim(f_0)) ==> f_0;

```

Figure 4.6: The extended FlashFill DSL; grammar rules only in the extended DSL are **bold**.

Grammar The DSL is given as a context-free grammar. Each line defines an option for a non-terminal given on the left of the `::=`. For most rules, the right side gives a DSL-defined function and a list of non-terminals for the arguments to that function. Functions are .NET functions defined in the class specified at the top of the file. The semantics of the DSL must be functional; that is, all functions called must be pure. Note that loops can be handled in a pure way by using lambdas. In general a while loop can be written using the function `WhileLoop(condition, body, final) = state => condition(state) ? WhileLoop(condition, body, final)(body(state)) : final(state)`.

Rules other than DSL-defined functions are written in all-caps starting with an underscore to distinguish them. These are used for a few different purposes. First, some are items that are not functions like constants, lambda variables, and lambda abstraction. Some reference items depend on the LaSy program: `_PARAM` corresponds to any parameter of the correct type and `_LASY_FN` allows for a call to another LaSy function. Those starting with a double underscore are for functions with specialized synthesis strategies. `__CONDITIONAL(b, e)` means that some number of `if...else if...else` branches are allowed where the conditions match the non-terminal `b` and the branches match the non-terminal `e`; this could be a DSL-defined function except for the fact that the synthesizer is aware of the semantics of conditionals and has specialized logic for learning them described in Section 4.4.2. Similarly, while not used in the example, `__FOR(i)` and `__FOREACH(arr)` have associated synthesis strategies for certain forms of loops that are described in Section 4.4.3.

Constant value generation The DSL may include code to decide what constant values may appear in the program which may depend on the examples (not shown in the figure). The simplest logic a DSL could use is that any values in the examples may be used as constants in the program. Other DSLs may have cases like including only regular expressions from a preset list that match one of the inputs or, when synthesizing XML, extracting the names of the tags and attributes in the outputs.

Rewrite rules The **rewrite** rules allow the DSL designer to express algebraic identities in their language to help prune the search space of programs with identical semantics.

4.3 Test-Driven Synthesis

The Test-Driven Synthesis methodology synthesizes a program by considering a sequence of examples in order and building up iteratively more complicated programs. We will describe the methodology for a LaSy program with one function, but it easily generalizes to multiple functions. Section 4.3.1 describes the algorithm in detail. Section 4.3.2 details how the iterative nature of the algorithm works. Section 4.3.3 discusses the importance of the order of examples.

4.3.1 Algorithm

The Test-Driven Synthesis algorithm (TDS in Algorithm 4.1) synthesizes a program P given a sequence of examples S and a set of base components. By “program” we mean a single function with a specified set of input parameters and return type. By “example” we mean a set of input values for those parameters and the correct output value. By “component” we mean any of the set of expressions known to the synthesizer which are used as the building blocks for the synthesized program; the base components are the functions referenced by the DSL in the LaSy program but components may also be partially filled-in function calls or larger DSL expressions.

In the spirit of TDD, we build P up, a little at a time, to allow synthesis of larger programs. P_i satisfies the first i examples; its successor program P_{i+1} is built by the DSL-Based Synthesis (DBS) algorithm using the first $i + 1$ examples along with information from P_i . The previous program P_i is used in three ways:

1. Its subexpressions are added to the component set.
2. *Contexts* to synthesize in are formed by removing each subexpression of P_i one at a time.

Algorithm 4.1: TDS(S, \mathcal{L})

input : sequence of examples S , DSL specification \mathcal{L}
output a program P that satisfies S or FAILURE
:

- 1 $e \leftarrow$ all grammar rules in \mathcal{L} ; $P_0 \leftarrow \perp$; failuresInARow \leftarrow 0;
- 2 **foreach** $i \leftarrow 0, \dots, |S| - 1$ **do**
- 3 **if** $P_i(\text{input}(S_i)) = \text{output}(S_i)$ **then**
- 4 $P_{i+1} \leftarrow P_i$;
- 5 failuresInARow \leftarrow 0;
- 6 **else**
- 7 contexts $\leftarrow \emptyset$, exprs $\leftarrow e \cup$ parameters of P_i ;
- 8 **foreach** subexpression s of P_i **do**
- 9 Add $\lambda \text{expr}. P_i[s/\text{expr}]$ to contexts;
- 10 Add s to exprs;
- 11 **foreach** branch B of P_i **do**
- 12 **foreach** subexpression s of B **do**
- 13 Add $\lambda \text{expr}. B[s/\text{expr}]$ to contexts;
- 14 $P_{i+1} \leftarrow$
DBS(contexts, (S_0, \dots, S_i) , exprs, \mathcal{L} , num_branch(P_i) + failuresInARow);
- 15 **if** P_{i+1} is *TIMEOUT* **then**
- 16 $P_{i+1} \leftarrow P_i$;
- 17 failuresInARow \leftarrow failuresInARow + 1;
- 18 **else**
- 19 failuresInARow \leftarrow 0;
- 20 **if** failuresInARow = 0 **then**
- 21 **return** $P_{|S|}$;
- 22 **else**
- 23 **return** FAILURE;

3. The number of branches may only exceed the number of branches in P_i if the previous example could not be satisfied (`failuresInARow > 0`). New conditionals are allowed only after failures in order to avoid overfitting to the examples by creating a separate branch for each one.

Example 1 (Walkthrough of TDS) We will use the DSL $C ::= \text{CharAt}(S, N) | \text{ToUpper}(C)$, $S ::= \text{Word}(S, N) | _PARAM$, $N ::= 0 | 1$ where $\text{Word}(s, n)$ selects the n^{th} word from the string s and $_PARAM$ is any function parameter and e is the set of all grammar rules in that DSL to demonstrate synthesizing the function $f(a) \Rightarrow \text{ToUpper}(\text{CharAt}(\text{Word}(a, 1), 0))$:

$i=0$: $S_0 = (a = \text{"Sam Smith"}, RET = 'S')$. $\text{exprs} = e \cup \{a\}$ (the whole language plus the parameter a) and $\text{contexts} = \{\circ\}$ (the set containing just the trivial context) because the previous program $P_0 = \perp$, so there are no subexpressions to remove to build contexts out of. The smallest program to compute 'S' is to select the first character of a , and therefore $P_1 = f(a) \Rightarrow \text{CharAt}(a, 0)$.

$i=1$: $S_1 = (a = \text{"Amy Smith"}, RET = 'S')$. $\text{contexts} = \{\circ, \text{CharAt}(\circ, 0), \text{CharAt}(a, \circ)\}$ because P_1 has two subexpressions that can be removed. exprs now also contains the expression $\text{CharAt}(a, 0)$. The simplest program consistent with both examples selects the second word of a instead of a itself, so DBS will generate $\text{Word}(a, 1)$ to select the second word and plug it into the second context to generate $P_2 = f(a) \Rightarrow \text{CharAt}(\text{Word}(a, 1), 0)$.

$i=2$: $S_2 = (a = \text{"jane doe"}, RET = 'D')$. $\text{exprs} = e \cup \{a, \text{Word}(a, 1), \text{CharAt}(\text{Word}(a, 1))\}$. $\text{contexts} = \{\circ, \text{CharAt}(\circ, 0), \text{CharAt}(\text{Word}(\circ, 1), 0), \text{CharAt}(\text{Word}(a, \circ), 0)\}$. Notably, $\text{CharAt}(a, 0)$ does not appear in exprs despite it appearing in exprs for the $i = 1$ step because it is not a subexpression of P_2 . It is important that such temporary diversions are forgotten so time is not wasted on them in later steps. DBS will output $P_3 = f(a) \Rightarrow \text{ToUpper}(\text{CharAt}(\text{Word}(a, 1), 0))$ which takes only a single step because it is the application of `ToUpper` to P_2 which appears in exprs .

We now discuss a few details of the algorithm to clarify the description and justify some design choices.

Relation between TDS and DBS DBS is described later in Section 4.4. We separate TDS from DBS both to show explicitly how the previous program P_i is used when constructing the next program P_{i+1} and to highlight the two key novel ideas in our approach:

1. TDS encapsulates the new idea of treating the examples as a sequence and using that fact to iteratively build up P by way of a series of programs which are correct for a subset of the input space defined by a prefix of the examples.
2. DBS encapsulates the parameterization by a DSL which allows for the flexibility of the algorithm.

TDS runs DBS repeatedly, each time giving it the next example from S along with expressions and contexts from the program synthesized in the previous iteration. In other words, it iteratively synthesizes P_k for each $k \leq |S|$ where P_k is synthesized using S_0, S_1, \dots, S_{k-1} ² and the previous program P_{k-1} . In this formulation, P_0 is the empty program \perp , or `throw new NotImplementedException();` in C#.

State As described, the only state kept between iterations is the program P_i and the failure count. DBS does not maintain state, and `contexts` and `exprs` depend only on P_i . Additionally, DBS is passed all of the examples up to S_i , not just S_i . One could imagine a more general problem definition where arbitrary (or at least more) state could be kept between invocations of DBS, but in our experience this tended to be more harmful than helpful: **preserving state essentially corresponds to not forgetting about failed attempts.**

² S_i is element i of the sequence S , 0-indexed

No lookahead Although we have formulated the problem as giving TDS the sequence of tests, notice it does not look beyond test S_i to generate P_{i+1} . Hence in an interactive setting the user could look at P_{i+1} or its output when choosing S_{i+1} .

4.3.2 Contexts and subexpressions

The intuition for the strategy of replacing subexpressions is that the program generated so far is doing the correct computation for some subset of the input space and is overspecialized to that subset. In the example above, after the $i = 0$ step, we had the program that returns the first character of the string instead of the first character of the second word of the string. That program was overspecialized to inputs where the first and second word start with same letter. Selecting the first letter was the right computation but on the wrong input, so filling in the context `CharAt(o, 0)` with the right input gave the desired program.

Each context represents a hypothesis about which part of the program is correct and correspondingly that the expression removed is overspecialized. Note that the expression appears in the set of components, so if a small change is sufficient, the effort to build it in previous iterations will not be wasted. Also, one such hypothesis is always that the entire program is wrong and should be replaced entirely.

Contexts are made out of each branch as well as the entire program in order to better support building new conditional structures (Section 4.4.2) using parts of one or more of the existing branches.

Pruning contexts While the algorithm as described uses every valid context that lies on a branch used by an unsatisfied example, the set of contexts could be pruned using ideas from fault localization research. Specifically, the idea is to use angelic debugging [7], which essentially means to put a symbolic value in each context and, for each example, ask a symbolic execution engine like Pex [57] whether any value exists that would satisfy that example. If there is no value for any given example, then there definitely isn't any expression

that would satisfy that example, so there is no point in trying expressions in that context.

There are two easy optimizations to this algorithm that greatly reduce the number of times Pex would need to be called. First, for examples that are already satisfied, we already know the answer is yes because we have the existing expression as a witness, so Pex only needs to be queried for unsatisfied examples. Second, contexts can be considered in top-down order of where in the AST they are removing a subtree. If Pex reports that in a given context, no value will satisfy a given test case, then the same will be true for any context formed by removing only a subexpression of the expression removed to form the original context.

Single hole This theory does not limit contexts to a single hole, but, empirically, doing so keeps the number of contexts manageable and seems to be sufficient in practice. Also, it allows the algorithm to prune away locations based on whether they are reached when executing a failing example: modifications elsewhere could not possibly affect whether such examples are handled correctly. If we allowed multiple modifications, the choice of modification points would have to be changed after any modification affecting control flow.

4.3.3 Example order

“The Transformation Priority Premise” [36] observes that in TDD the test case order can affect the ability of the programmer to produce a program through small code changes. Similarly, our algorithm may fail to synthesize a program if not given examples in a good order—after all, one of our key insights is that the ordering of examples is a useful input to the synthesizer. We claim that a good example order corresponds to the natural concept of examples of increasing complexity, but we do not have evidence to back up this assertion.

As a “good” order is defined as being one that results in synthesizing a program satisfying the specification the user has in mind, it is unclear how to define a “good” order without referencing the final synthesized program. Needless to say, such a definition cannot be directly used to guide the generation of a sequence of examples. This is unsatisfying, but we

provide some intuition on what such orders look like and Section 4.5.2 gives evidence that our synthesizer is robust to small variations in the order of examples. We thus remark that a human could learn to produce such an ordering just like a human can learn to produce TDD test cases in an order that easily results in a correct program. Fundamentally, this is analogous to the issue of how a human should generate a concise but sufficient set of black-box tests for a program. Many guidelines exist, but there is no precise methodology. Nonetheless, black-box testing is successful.

Once the user has covered the entire intended specification, the user has produced a suite of simple test cases for the algorithm being synthesized. As in TDD, to confirm that the synthesized procedure is correct, the user should write a few larger, more comprehensive tests of the procedure.

4.4 *DSL-Based Synthesis*

The DSL-Based Synthesis algorithm (DBS in Algorithm 4.2) is the part of TDS that actually generates new programs. DBS takes as input a set of examples S , a set of contexts C which generated DSL expressions are plugged into to form synthesized programs matching \mathcal{L} , a set of expressions e , a DSL definition \mathcal{L} , and a maximum number of branches m . It outputs a new program P' that satisfies all examples in S or `TIMEOUT` if it is unable to do so.

The algorithm is built on five key concepts:

1. New programs are not generated directly; instead expressions are generated and plugged into contexts provided as hints to narrow the search space. This is used by TDS to indirectly provide the previous program as a hint (Section 4.3.2).
2. New expressions are formed from all compositions of expressions according to the DSL \mathcal{L} . To produce all smaller expressions before generating larger ones, DBS runs as a series of iterations, where, in each iteration, only expressions from previous iterations are composed into new expressions. Section 4.4.1 discusses generation of new expressions (and important optimizations).

Algorithm 4.2: DBS(C, S, e, \mathcal{L}, m)

```

input : set of contexts  $C$ , set of examples  $S$ , set of expressions  $e$  to build new
         expressions from, DSL specification  $\mathcal{L}$ , maximum number of branches  $m$ 
output a program  $P'$  that satisfies  $S$  or TIMEOUT
:
/* Try generates one or more programs and if one satisfies  $S$ , DBS
   returns it. */
1 Try loop strategies in a separate thread (Section 4.4.3);
2  $allExprs \leftarrow e$ ;
3 while not timed out do
4   foreach  $c \leftarrow C$  do
5     foreach  $expr \leftarrow allExprs$  do
6       Try  $c(expr)$ ;
7   Try conditional solutions up to  $m$  branches (Section 4.4.2);
8    $allExprs \leftarrow$  generate new expressions (Section 4.4.1);
9 return TIMEOUT;

```

3. A new branching structure will be synthesized if no generated program satisfies all examples in S and $m > 1$. Only programs containing at most m branches will be synthesized in order to avoid over-specializing to the examples. Synthesis of conditionals is discussed in detail in Section 4.4.2.
4. If the algorithm times out before a solution is found, it will return a special failure value TIMEOUT. In TDS, this case increments m allowing for more branches in the next run of DBS.
5. The search space can be reduced even further using specialized strategies for some functions. We demonstrate this by describing strategies we defined for a couple common loop forms in Section 4.4.3.

4.4.1 *Choosing new expressions*

New expressions to use in the contexts are generated by component-based synthesis [25]. In component-based synthesis, a set of components (expressions and methods) are provided as input and iteratively combined to produce expressions in order of increasing size until an expression is generated that matches the specification. In our case, the “specification” is the examples. As opposed to previous component-based synthesis work, the generation of new expressions is guided by a DSL \mathcal{L} and instead of testing the expressions against the specification, they are used to fill in contexts producing larger programs which are then tested.

In our system, all components are expressions marked with which non-terminal in the grammar defined them. Methods are represented as curried functions. The synthesizer generates new expressions by taking one curried function and applying it to an expression marked with the correct non-terminal. Each iteration of the synthesizer does so for every valid combination of previously generated expressions in order to generate programs of increasing size. Representing methods as anonymous functions also simplifies handling methods that themselves take functions as arguments, which are common in higher-order functions like `map` and `fold`.

As the number of components generated after k iterations is exponential with the base being the number of grammar rules (i.e., functions and constants in the DSL) in the worst case, a DSL that is too large will cause DBS to run out of time or memory before finding a solution. In practice, around 40–50 grammar rules seems to be the limit for DBS, but it depends greatly on the structure of the DSL. An earlier version of DBS without the optimizations described below could not handle more than around 20–30 grammar rules. Further optimizations to better prune the search space or reduce the number of contexts could possibly allow for even larger DSLs.

Optimizations

Minimizing the number of generated expressions is important for performance. Redundant expressions are eliminated in two ways: the first is syntactic and hence it is fast and always valid, while the second is semantic and valid only when an expression does not take on multiple values in a single execution (e.g., if the program is recursive).

Syntactic All expressions constructed are rewritten into canonical forms according to the **rewrite** rules in the DSL and duplicates are discarded. For example, $x+y$ and $y+x$ are written differently but can be rewritten into the same form so one will be discarded. DBS will only accept sets of **rewrite** rules which are acyclic (once commutativity and other easily broken cycles are removed) to ensure there is a canonical form. Related to this, constant folding is applied where possible, so, for example, $2*5$ and $5+5$ would both be constant folded to 10, further reducing the search space.

Semantic The vast majority of the time, an expression takes on only a single value for each example input. In other words, the expression is equivalent to a lookup table from the example being executed to its value on that example. Only expressions with distinct values are interesting, so, for example $x*x$ and $2+x$ would be considered identical if the only example inputs were $x = 2$ and $x = -1$. This is similar to the redundant expression elimination in version space algebras [29]. The exceptions are if the expression is part of a recursive program or lambda expression, in which case this optimization is not used because the expression will be evaluated multiple times for each example input.

4.4.2 Conditionals

So far we have not considered synthesizing programs containing conditionals, which are of course necessary for most programs. We consider first synthesizing programs where a single cascading sequence of **if...else if...else** expressions occur at the top-level of the function body, with each branch not containing conditionals. Then the goal is to have as

few branches in the one top-level conditional as possible. The problem is to partition the examples into which-branch-handles-them to achieve this goal.

For every program p DBS tries, the set of examples it handles correctly is recorded and called $T(p)$. If $T(p) = S$ (all examples handled), p is a correct solution and can be returned. Otherwise, each set of programs Q (where $|Q| \leq m$) whose union of handled examples $\bigcup_{p \in Q} T(p)$ equals S is a candidate for a solution with appropriate conditionals. To be a solution, Q also needs guards that lead examples to a branch that is valid for them; to simplify this, whenever a boolean expression g is generated, the set of examples it returns true for, $B(g)$, is recorded. The sets Q are considered in order of increasing size, so if there are multiple solutions, the one with the fewest branches will be chosen.

If the conditional does not appear at the top-level, then it must appear as the argument to some function. To handle this case, we note that if every branch of the conditional generated as described already happens to contain a call to a function f with different arguments, then it could be rewritten such that the call to f occurs outside of the conditional if that is allowed by the DSL. In that case, we can say that all of the branches match the context $f(\circ)$.

In the algorithm, for each non-terminal the DSL allows for conditionals at, each program p is put into zero or more buckets labeled with the context that non-terminal appears in. For example, if the argument of f may be a conditional and $p = f(f(x))$ then p would be put in the buckets for $f(\circ)$ and $f(f(\circ))$. Then the same algorithm as above is run for each bucket with the conditionals being rewritten to appear inside the context. Inserting multiple conditionals just involves following this logic multiple times.

4.4.3 *Loops*

The primary way DBS handles loops is to simply not do anything special at all: recursion and calling higher-order functions like `map` and `fold` are handled by the algorithm as described so far. As described in Section 4.2.2, a general `WhileLoop` higher-order function can be used to express arbitrary loops that DBS may synthesize like any other DSL-defined function.

Loops violate the assumption made by our optimizations that expressions will only ever

```

loop strategy "for" ex exs:
  choose param "i" of type I;
  let exs' = exs.Where(ex' => ex.Except("i") == ex'.Except("i"));
  newparam "acc" = exs'.Single(ex' => ex'["i"]+1 == ex["i"])._OUTPUT;
  return = ex._OUTPUT;
  loop_var "i" I:
    initial = exs'.Min(ex' => ex'["i"]);
    computation = i + 1;
  loop_var "acc" *:
    initial = exs.ArgMin(otherEx => otherEx["i"])._OUTPUT;
    computation = _BODY;
  loop_condition = i < ex["i"];
  loop_output = acc;

loop strategy "foreach" ex exs:
  choose param "arr" of type A;
  select newparam "i" from [0..ex["arr"].Length-1];
  newparam "current" = ex["arr"][ex["i"]];
  newparam "acc" = ex._OUTPUT.Take(ex["i"]);
  return = ex._OUTPUT[ex["i"]];
  loop_var "i" I:
    initial = 0;
    computation = i + 1;
  loop_var "acc" A:
    initial = EmptyIntArray;
    computation = Append(acc, _BODY);
  loop_condition = i < ex["arr"].Length;
  loop_output = acc;

```

Figure 4.7: Example loop strategies

take on a single value throughout the execution of the program. We can detect that and disable the optimization, but we can also notice that loops act in a very structured manner that can be encoded into the algorithm. The algorithm will not handle all loops that could ever be written, but instead will be specialized for a few common cases like iterating all values of an input array in order or iterating over the integers from some number (usually 0 or 1) to an input variable. For the application of handling loops that appear in small simple programs, this is often sufficient.

This section discusses two such common patterns we have written strategies for; experts designing DSLs may additionally define their own strategies for other forms of loops. These can be used in a DSL via the `__FOREACH(E)` or `__FOR(E)` rules where **E** is the non-terminal for the body of the loop.

For loop example

The way the algorithm handles loops is by adding a separate pre-processing step. The observation is that if the return value is computed in the loop, then with enough test cases, the behavior of the loop body is visible in the test cases. The simplest example is a loop

```
public static int ForLoopExample(int n) {
    int res = 0;
    for(int i = 0; i < n; i++) {
        res += i;
    }
    return res;
}
```

The test case for, say, `n=3` tells us both the result when `n=3` and the value of `res` at the start of the `i=3` iteration of the loop. We essentially rewrite the code as

```
res = LoopBody(res, i, n);
```

and synthesize `LoopBody()`, which does not need to have any mutation in it.

Given the examples `ForLoopExample(3)=6`, `ForLoopExample(4)=10`, we can construct the example `LoopBody(6, 4, 4)=10` by taking the output for the `n=3` example as the first input and synthesize `res+i` for the code of `LoopBody()`, which will result in synthesized code that looks like the example code we started with.

Foreach loop example

The “foreach” loop strategy’s hypothesis is that there is a 1-to-1 correspondence between an input array and an output array. Assuming that hypothesis, the examples can be split into one example for each element where `i` is the index, `current` is the element at that index, and `acc` is the array of outputs for previous indexes:

in	RET
{3, 5, 4}	{9,25,16}

would become the examples

in	i	current	acc	RET
{3, 5, 4}	0	3	{}	9
{3, 5, 4}	1	5	{9}	25
{3, 5, 4}	2	4	{9,25}	16

(The **select newparam** rule in Figure 4.7 allows for creating multiple examples from a single example.) Those examples could be used to synthesize the loop body `current*current` using TDS. The strategy includes the boilerplate code to take the loop body `current*current` and output a **foreach** loop over the input array.

That example is overly simple as such a computation could easily be captured by a `map`. However, loop strategies also allow for loops that are not as easily expressed with higher-order functions. For example, the loop bodies examples could also include the values computed so far:

in	RET
{5, 2, 3}	{5,7,10}

would become

in	i	current	acc	RET
{5, 2, 3}	0	5	{}	5
{5, 2, 3}	1	2	{5}	7
{5, 2, 3}	2	3	{5,7}	10

Then the synthesized loop body would be `acc.Length > 0 ? current + acc.Last() : current`, which could be rewritten into a loop computing the cumulative sum of `in`:

```
int[] res = new int[in.Length];
for(int i = 0; i < in.Length; i++) {
    int current = in[i];
    int[] acc = res.Take(i).ToArray();
    // This line is synthesized, the rest is boilerplate.
    res[i] = acc.Length > 0 ? current + acc.Last() : current;
}
return res;
```

Loop strategies

Figure 4.7 shows the two loop strategies described in pseudocode.

A loop strategy is a set of primarily syntactic rules that go along with a hypothesis about the looping structure of the program to synthesize. Given that hypothesis, the first set of rules (**choose**, **select newparam, newparam, return**) take the examples and produce a set of examples for the loop body that will include as new parameters the loop's temporary variables. The **loop_*** rules build a loop using the synthesized loop body. If the hypothesis holds, then that loop will satisfy the original examples.

Different loop strategies can give different information like including the index in a **foreach** or giving `acc` corresponding to going in reverse order. Furthermore, the concept of splitting up arrays by element to find patterns can also be applied to splitting strings (by

length or delimiters), XML nodes, or whatever other structured data may be in the target domain.

4.4.4 *Conditionals and loops, a general theory*

DSL definitions contain rules with and without specialized strategies. Most rules, including all DSL-defined functions, do not have specialized strategies so expressions using those rules are built using the default strategy of searching through the semantically distinct expressions (using the example inputs to decide which expressions are distinguishable). On other hand, we have defined strategies for conditionals and loops that use the example outputs as well as the inputs to power more directed approaches to learning those constructs. While we referenced the output values directly in the explanation of the strategies for loops, the discussion of conditionals only referenced them indirectly by keeping track of which examples a program was correct for.

The strategies for conditionals and loops should be considered as just different instances of the same concept. While conditionals merely select a subset of the examples for each branch, loops do larger rewrites of the examples used in the recursive calls to the synthesizer. Theoretically, a DSL designer could include other strategies like inverses of DSL-defined functions or a polynomial solver for synthesizing arithmetic. We presently omit such functionality because we believe it places undue burden on the DSL designer.

4.5 *Evaluation*

We would like to demonstrate that order of examples is an easy input for a human to provide and that using it as the input of our program synthesizer allowed us to easily develop an algorithm that is fast, effective, and domain-agnostic. Due to not performing a user study, we do not have evidence that humans can easily provide an order of examples that works well for our system. We do show that our system works well across multiple domains using examples from help forums written by users, but none of example sequences are long enough that the ordering is being used (although we show the iterative nature of the algorithm

is helping). We do show that the system is using order of examples for more difficult to synthesize programs, but those examples were written by us, not users in a user study. Also, we do have evidence that our algorithm is fast, effective, and domain-agnostic.

Our evaluation demonstrates that TDS is sufficiently powerful and general to synthesize non-trivial—albeit small—programs in multiple domains from small sequences of real world examples obtained from help forums. We also compare TDS to Sketch [52], the present state-of-the-art in domain-agnostic program synthesis, and to state-of-the-art specialized synthesizers where applicable.

Furthermore, we explored how sensitive our iterative synthesis technique actually is to the precise ordering of examples, showing that example order is significant to speed or ability to synthesize on a non-trivial proportion of the examples, especially on larger programs. We also validated some of our design decisions by selectively disabling parts of our algorithm.

All experiments were run on a machine with a quad core Intel Xeon W3520 2.66GHz processor and 6GB of RAM.

Section 4.5.1 describes the benchmarks used in our experiment and our success in synthesizing them and compares TDS to prior specialized synthesizers where applicable. Section 4.5.2 investigates the effect of example ordering to our synthesizer’s performance. Section 4.5.3 breaks down the usefulness of the different parts of our algorithm. Section 4.5.4 evaluates performance for our synthesizer and validates our timeout choice.

4.5.1 *Benchmarks*

We evaluate our technique in four domains: String transformations compares our technique to a state-of-the-art specialized programming-by-example system for string transformations, Table transformations compares our technique to a start-of-the-art specialized programming-by-example system for table transformations, XML transformations discusses using our system for the novel domain of XML transformations, while finally Pex4Fun programming game shows our technique is able to automate coding in TDD for introductory programming problems, which gives us difficult enough programs that order of examples comes into play.

For the first three, the example sequences used and output of our synthesizer can be found at <https://homes.cs.washington.edu/~perelman/publications/pldi14-tds.zip> and a by-program result summary can be found in Table 4.1; the last section’s programs are not public.

String transformations

String transformation programs take as input one or more strings and output a string constructed from the input using mainly substring and concatenation operations.

Strings are a natural format for input/output examples that often appear in real-world end-user programming tasks as recent work by Gulwani et al. [15] has shown: their work became the FlashFill feature in Excel 2013 [2]. Unlike FlashFill, TDS does not use careful reasoning about the domain of strings, but it is still able to quickly synthesize many of the same examples as well as some similar programs that the prior work (i.e., FlashFill) cannot synthesize.

Benchmarks To compare against FlashFill, we first defined exactly the FlashFill DSL in our DSL definition language and ran our synthesizer on the examples that appear in the FlashFill paper ([15]) to confirm we could synthesize them. Then we made a few modifications to the DSL which are shown in Figure 4.6 to make it more general; specifically, we allowed nested substring operations, substring indices dependent on the loop variable, and calls to other LaSy functions.

In addition to WordWrap and the examples from the FlashFill paper, we wrote test case sequences for 7 simple real world string manipulation examples outside of the scope of FlashFill but handled by our extended DSL including selecting the two digit year from a date (requires nested substrings), reversing a string (requires substring indexes dependent on the loop variable), and bibliography examples like the one in Figure 4.2 (requires a user-defined lookup).

category	benchmark	used	given	time	grammar	# b
string	ext-alternate-chars	1	1	1.4	FF-extended	1
string	ext-bibliography	8	8	15.2	FF-extended-no-loops	3
string	ext-extract-year2	2	4	3.2	FF-extended	1
string	ext-middle-initial	1	2	19.2	FF-extended	1
string	ext-reverse	2	3	0.3	FF-extended	1
string	ext-start-of-third	3	3	0.2	FF-extended	1
string	ext-wordwrap	24	24	23.2	FF-extended-recursive	3
string	popl11-ex01	2	3	19.4	FF-no-loops	1
string	popl11-ex02	2	5	4.3	FF	1
string	popl11-ex03	2	2	2.2	FF	1
string	popl11-ex04	2	3	0.1	FF-reduced	1
string	popl11-ex07	6	6	0.4	FF	3
string	popl11-ex08	5	5	0.2	FF-reduced	3
string	popl11-ex09	6	6	1.2	FF-no-loops	3
string	popl11-ex10	5	5	2.6	FF-no-loops	2
table	CopyRows-small	2	2	6.5	TableTransform	1
table	ext-repeated-column1	2	2	2.9	TableTransform	1
table	ext-repeated-column2	2	2	29.3	TableTransform	1
table	ext-repeated-column3	2	2	60.0	TableTransform	1
table	pldi11-fig01	6	6	7.2	TableTransform	1
table	pldi11-fig07	2	2	21.1	TableTransform	1
table	pldi11-fig08	1	1	5.9	TableTransform	1
table	pldi11-fig10	3	3	5.4	TableTransform	1
XML	books-sorted	3	3	7.0	XML	1
XML	books	3	3	4.9	XML	1
XML	example03	1	1	6.2	XML	1
XML	example04	2	2	7.2	XML	1
XML	example05	2	2	1.9	XML	1
XML	extractTitle	1	1	11.2	XML	1
XML	filterElements	1	1	2.1	XML	1
XML	flattenAuthorInfo	1	1	18.5	XML	1
XML	numberElements	1	1	7.2	XML	1
XML	sortElements	1	1	4.1	XML	1

Table 4.1: End-user data transformation benchmark results. “used” is the number of examples used by the synthesizer while “given” is the total number that appeared in the input, which could be greater if the synthesizer got the right answer before using all of the examples. “time” is the computation time in seconds. “# b” is the number of branches in the final program. “FF” is short for “FlashFill”.

Results Each of the 15 example sequences contains 1–8 examples except for word wrap which uses 24 examples. 5 of the examples can be synthesized in under a second. 6 take more than 1 second but under 5 seconds, while the other 4 finish in under 25 seconds. FlashFill synthesizes all of the examples it can handle in well under a second. Simply by specifying the DSL, our domain-agnostic synthesis technique approaches the performance of a state-of-the-art specialized synthesis technique while maintaining the ability to generate more complicated control flow structures.

We coded all examples using the corresponding DSLs in Sketch and none of them completed within 10 minutes.

Table transformations

Table transformations convert spreadsheet tables between different formats by rearranging and copying a table’s cells.

Benchmarks Harris et al. [20] give a DSL and synthesis algorithm for these transformations along with a collection of benchmarks the authors found on online help forums. We defined their DSL for our synthesizer and ran it on their benchmarks.

For additional benchmarks not handled by their DSL, we added more predicates to the grammar to allow it to handle a wider range of real world normalization scenarios. For example, our extended grammar can support converting various non-standard spreadsheets with subheaders into normalized relational tables.

Results Each of the 8 benchmarks uses 1–6 examples. TDS synthesizes most of them in under 10 seconds; 2 take 30 seconds and one takes a full minute.

Their paper says Sketch was unable to synthesize their benchmarks so we did not attempt to run the benchmarks using Sketch.

XML transformations

XML transformations involve some combination of modifying an XML tree structure and tag attributes and string transformations on the content of the XML nodes.

Benchmarks We selected 10 different real world examples from online help forums and constructed a DSL able to express the operations necessary to synthesize programs for all of them. Two of the examples appear in Section 4.1.2.

One transformation involved putting a tag around every word, even when words had tags within them, which was easiest to express treating the words as strings instead of as XML. Making the string and XML DSLs work together required simply putting the functions to convert between the two in the DSL. This kind of cross-domain computation shows the strength of our domain-agnostic approach.

Results Most of the benchmarks used a single example while the rest used no more than 3. TDS synthesizes all but two of the benchmarks in under 10 seconds and the remaining two in under 20 seconds.

We also implemented the DSL and benchmarks in Sketch, which was unable to synthesize any of them within 10 minutes.

Pex4Fun programming game

Motivation Although we believe our end-user programming scenarios are compelling, we wanted to test our synthesizer in a scenario closer to the TDD programming style it was inspired by and get a source for some more challenging functions to synthesize, particularly ones that would show off the use of order of examples.

To that end, we had our synthesizer play the Pex4Fun [55] programming game where a player is challenged to implement an unknown function given only a few examples. For each attempt, the Pex [57] test generation tool uses dynamic symbolic execution to compare the player’s code to a secret reference solution and generates a distinguishing input if the

player’s code does not match the specification. Pex provides the test for the test step of the TDD while the player is performing the programming step without full knowledge of the specification. This is a more “pure” form of TDD as the player’s coding is not biased by knowing the specification, giving a closer parallel to our synthesizer which does not have knowledge of the specification of the function it is synthesizing.

Experiment description We use a single DSL with a set of 40 simple `string` and `int` functions that may be combined in any type-safe way to show that while a carefully constructed DSL can be given to our synthesizer to make it perform especially well in a given domain, it can still successfully synthesize programs using a less specialized DSL capable of describing a wider range of programs. Note that our DSL was written without looking at the Pex4Fun puzzles and therefore ended up missing some functions necessary for some puzzles like bitwise operations.

For each puzzle in the Pex4Fun data, we had our algorithm play Pex4Fun for a maximum of 7 iterations, after which the synthesizer was considered to have failed if it still had not produced a solution. This may seem like too few iterations, but it is more calls to Pex than most users used for any of the puzzles.

For some of the puzzles, using Pex to generate test cases failed to generate a solution despite manual inspection determining that the puzzles were well within the capabilities of the synthesizer and the DSL used. In such cases, a sequence of test cases was generated manually to synthesize solutions to those puzzles.

Benchmarks The puzzles our synthesizer could synthesize included, among many others, factorial, swapping elements of an array, and summing numbers in a string delimited by a delimiter specified on the first line of the string. One more trivial example was concatenating the first and last element of a string array.

The remaining unsynthesized puzzles either involved looping structures not covered by

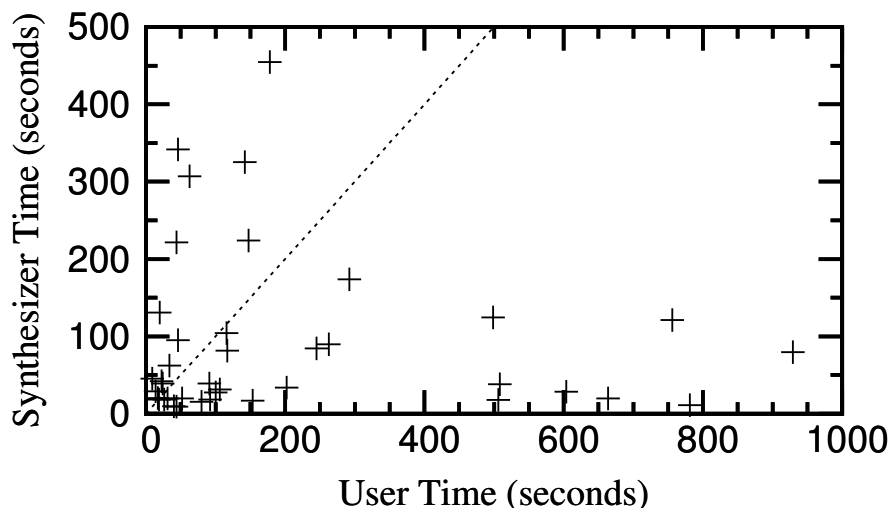


Figure 4.8: Average time spent by user compared to time spent by our synthesizer (including time waiting for Pex in both cases). Each data point is one puzzle. The dotted line is $y = x$.

our strategies (e.g., count the number of steps of the $3n + 1$ problem³ needed to reach 1), components not in our component set (e.g., compute bitwise or), or arithmetic expressions too large to construct using component-based synthesis (e.g., compute the value of a specific cubic polynomial).

Results We ran our experiment across 172 randomly selected puzzles from Pex4Fun. The synthesizer found solutions for 72 of those. For 60 of those, the test cases generated by Pex were sufficient, but for another 12 the test cases had to be written manually.

Comparison to humans Figure 4.8 and Figure 4.9 compare TDS playing Pex4Fun against the average of a randomly selected sample of users playing Pex4Fun on the same puzzles. Without reading into this data too much, it still has the high-level take-away that for puzzles TDS succeeds at, its performance is comparable to users of Pex4Fun. Figure 4.8 suggests TDS is not unreasonably slow (in fact, the chart shows that **for the puzzles it solves, TDS tends to solve them faster than the average user**). Figure 4.9 shows **TDS does not**

³https://en.wikipedia.org/wiki/Collatz_conjecture

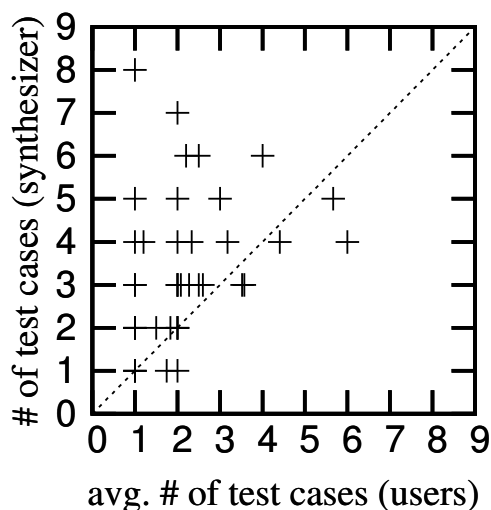


Figure 4.9: Average # tests used by user vs. TDS (distinct, including tests that pass when generated). Points are puzzles. Dotted line is $y = x$.

require significantly more information (measured in test cases) than the average user to solve a Pex4Fun puzzle, which was the inspiration for using it for the hint system for Code Hunt (the successor to Pex4Fun) described in Chapter 5. The y -axis of the chart goes above 7 because more than one test case may be added from Pex on each iteration if Pex returns any passing tests.

4.5.2 Example ordering

We hypothesized that example ordering is useful and TDS is robust to small variations in example order. For the vast majority of our benchmarks, the number of examples is small and their order is unimportant; the use of contexts and subexpressions from the previous program is important (when not synthesizing from a single example) as is shown in Section 4.5.3 by disabling them, but the specific order of the examples is not. On the other hand, the 12 Pex4Fun puzzles that required manually written example sequences were difficult enough for TDS that the ordering of examples was in fact valuable to the algorithm. To give an idea of how important the ordering actually was, we ran TDS on randomly reordered copies of those

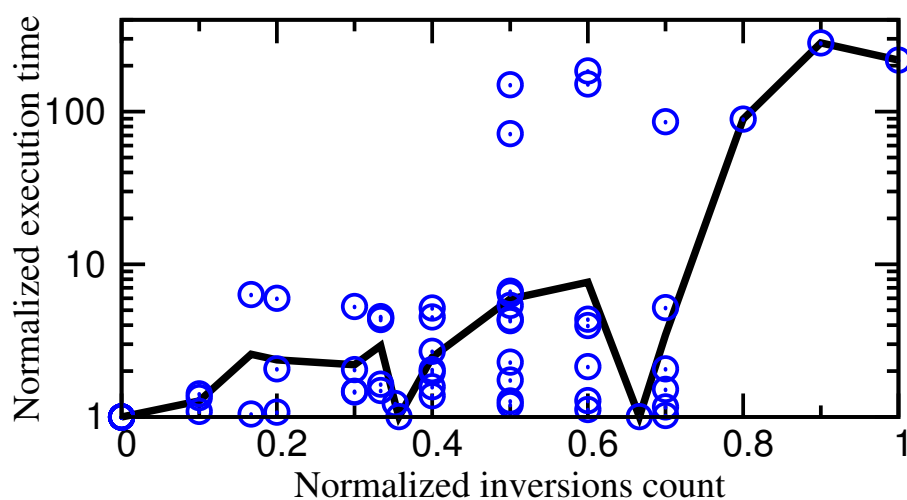


Figure 4.10: Normalized time (1=opt) for many reorderings of examples (distance measured in inversions, normalized so 1=reverse); the line shows the geometric mean of the data points for each normalized inversion count.

example sequences.

Figure 4.10 shows the timing results for the example sequences that were successfully synthesized. Each circle is one example sequence, and the line shows geometric mean of circles.

The x -coordinate measures how far from the optimal example sequence it was where 0 is the optimal sequence and 1 is the reverse of the optimal sequence. Specifically, the value is the number of inversions⁴ between the two sequences divided by the maximum possible number of inversions ($\frac{n(n-1)}{2}$ for a sequence of length n). The y -coordinate is the time it took to synthesize the solution using the random sequence normalized to the time it took to synthesize using the optimal sequence. Note that the y -axis is a log scale.

Figure 4.11 shows for how many of the reorderings the program was not successfully synthesized. The x -axis is the same; each bar corresponds to a range of normalized inversion counts. The bar heights are the proportion of sequences for which a program could not be synthesized, and the numbers above the bars are the absolute counts. There are more

⁴# of example pairs that have different order in the two lists

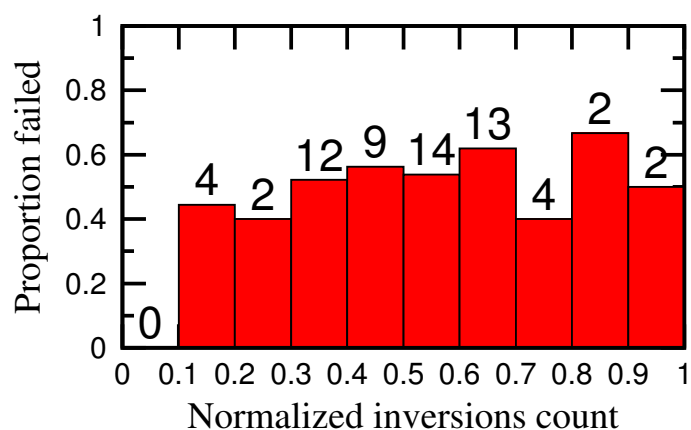


Figure 4.11: Proportion of reorderings TDS failed on by normalized inversion count. Numbers above bars are absolute # of failed reorderings.

examples toward the middle as the sequences were selected uniformly at random, and there are more possible sequences in the middle.

The charts show two important properties of TDS. **First, it does in fact depend on example ordering:** large changes to the example ordering make it take much longer to find a solution or fail to find a solution at all. This shows that we are, as claimed, getting value out of the order of examples. **Second, it is robust to small changes in the example ordering:** for normalized inversion counts less than 0.3, fewer than half the reorderings failed and those that succeeded took on average less than three times as long as the optimal ordering. This lends some experimental evidence to the assertion that humans can actually provide example orderings that are useful to the algorithm: if it were not robust to small changes, that would mean that useful orderings would be difficult to produce.

Those 12 examples show the worst case for our synthesizer. For the other 60 Pex4Fun puzzles with test cases sequences from Pex, 51 of them were also successfully synthesized with those test cases in reverse order. For the rest of the examples, nearly all could be synthesized with the examples sequence in reverse order, at worst slowed down by a factor of 3. This indicates that the sensitivity to test case ordering is affected by how complicated the program being synthesized is.

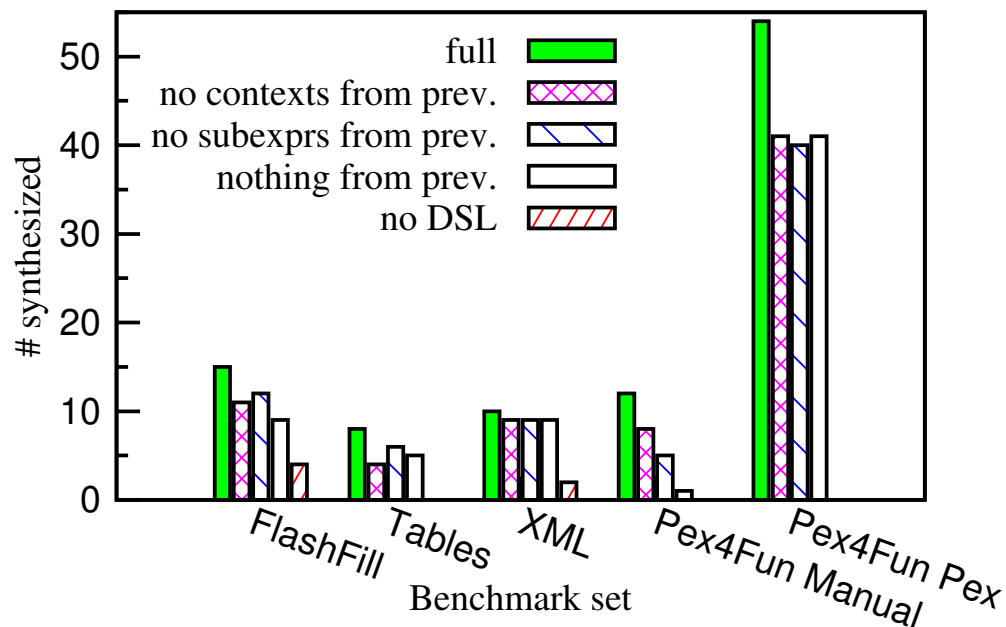


Figure 4.12: # synthesized by benchmark set and features enabled in algorithm. “full” is the full algorithm, contexts and subexpressions from previous program together form the information TDS uses.

4.5.3 Significance of various parts of algorithm

In order to evaluate the usefulness of the different parts of our algorithm, we ran our benchmarks with parts of it disabled. Figure 4.12 shows how many of the benchmarks were synthesized under each limited version of the algorithm.

Iterative synthesis The iterative synthesis strategy employed by TDS is implemented by passing contexts and subexpressions from the previous program to DBS. We disabled these two pieces of information individually and together. The Pex4Fun and table transformation benchmarks were most affected by the removal of features from TDS due to working with larger programs. Notably, we see that **either the subexpressions or contexts alone is helpful, but when combined they are significantly more powerful.**

Further, we stress that the iterative synthesis element of the approach is not the same as the example ordering. While the example ordering led to the iterative synthesis design

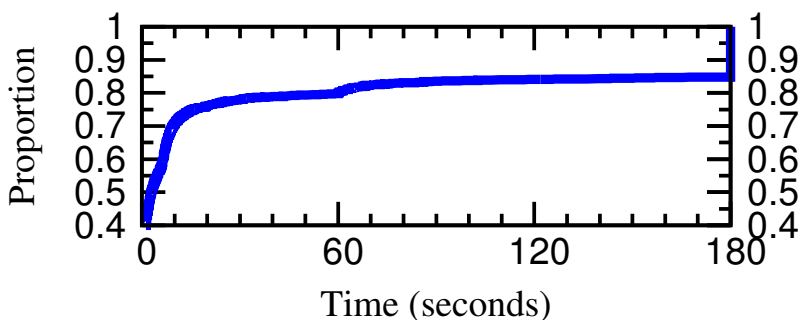


Figure 4.13: Execution times of all DBS runs.

and the iterative synthesis design is how the algorithm is able to take advantage of example ordering, being dependent on iterative synthesis for good results does not mean being dependent on example ordering for good results. As shown above, even for programs where example order is important, about half of the time a completely different order still works.

DSL-based synthesis We also disabled the use of the DSL when generating components in DBS, so it instead would be limited only by the types of expressions. There are no “no DSL” bars for the Pex4Fun benchmarks because the Pex4Fun DSL already only used the types, so that configuration is identical to the “full” configuration. Many of the other programs were synthesized from just a single example, so the weakening of TDS did not have a large effect, but this success was achieved due to the power DBS gained from the DSL as can be seen from the fact that very few of the end-user benchmarks could be synthesized without access to the DSL.

4.5.4 Performance

Figure 4.13 shows a CDF of all execution times of all of the DBS runs used in our experiments. **This chart shows that DBS is quite efficient with a median running time of approximately 2 seconds and running in under 10 seconds around 75% of the time.**

Timeout Throughout the experiments, we used a 3 minute timeout. Only very rarely in our experiments did DBS ever run for anywhere near 3 minutes without timing out. There is a visible bump around 60–70 seconds after which the line is almost flat, indicating that it is very unlikely that giving DBS a small amount of additional time would have made any noticeable difference in our results. This is further verified by *ad hoc* experience that without a timeout, DBS runs for over 30 minutes without a result or runs out of memory.

4.6 Conclusion

In order to develop a new programming-by-example algorithm, we used the natural human inclination to explain tasks with examples of increasing complexity and developed an algorithm that takes advantage of such sequences of examples. Using this information a human can easily provide to a programming-by-example system, our algorithm can run quickly and effectively across multiple domains with significantly less effort necessary for specializing to new domains than for other techniques. We demonstrated this by synthesizing programs across multiple domains with example sequences of varying lengths and altered both the sequences and what information our synthesizer uses from the sequences to demonstrate it uses order of examples effectively.

Chapter 5

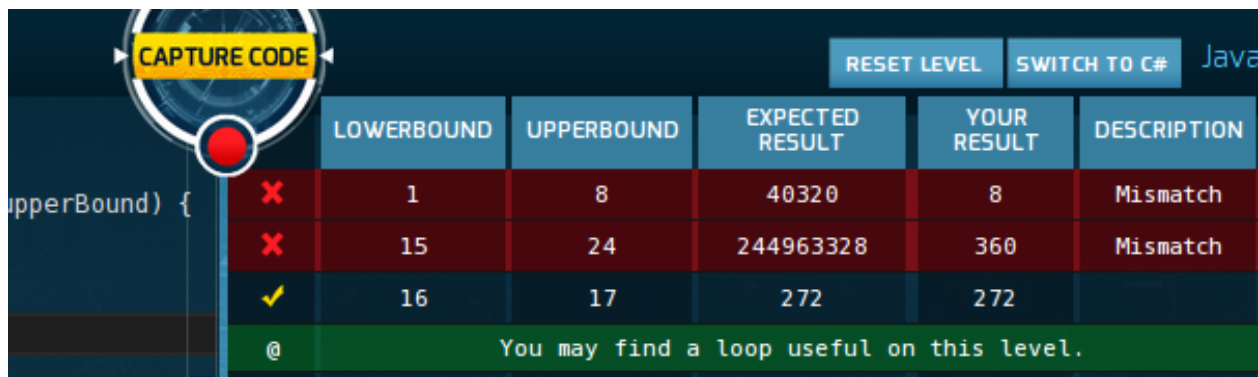
CODE HUNT HINT SYSTEM

The goal of this chapter is to develop an automated feedback system for small programming assignments at the scale of those seen in introductory computer science courses. We will be working in the context of an educational game, so our feedback will take the form of hints that will be displayed to the player (as opposed to information used to aid a human grader).

The game we are working on is Code Hunt [56], mentioned in Section 2.4.3, which is the successor to Pex4Fun [55], which was used in the Pex4Fun programming game benchmark in Chapter 4. As a reminder, the basic game play is that the player is trying to write a program to implement some secret functionality represented by a secret challenge solution provided by the creator of the level. After each attempt the player submits, the game responds with a set of inputs to the program along with the output of that attempt as well as the correct output. At least one of these will always be a failing test case until the player wins the level by writing a program that is functionally equivalent to the secret program.

The game has hundreds of levels and additionally allows any user to submit additional levels, so we cannot feasibly require per-level descriptions of how the system should generate hints. Given our design philosophy of asking what information the user can provide easily, we notice there is a large amount of information on each level available for no effort to the level creator: all of the attempts of all of the players on that level. Therefore, to minimize user effort, we will generate hints using just the attempts of other players on the same level.

Specifically, we mine from that data which expressions are useful on a given level and which are not useful. For a player who is close to a solution, this is enough to use the TDS program synthesizer described in Chapter 4 to find a solution similar to the player's attempt



		LOWERBOUND	UPPERBOUND	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
upperBound) {	✗	1	8	40320	8	Mismatch
	✗	15	24	244963328	360	Mismatch
	✓	16	17	272	272	
@	You may find a loop useful on this level.					

Figure 5.1: The Code Hunt test results screen.

and use the difference between the two programs to generate a hint that we call a “line hint” due to it specifying a line to change, similar to AutoGrader [51] (see Section 2.4.2). When a player is not close to a solution, this technique is not meaningful to apply, so instead we use the same data to nudge the player in the direction of a solution by using statistics to determine which expression is most worth recommending they use or avoid, which we call a “recommendation hint”.

5.1 The Code Hunt game

Code Hunt [56] is an educational programming game where the player writes a program in a standard program language (Java or C#) in a simple in-browser programming environment, but the hook is that the player is not told the specification of the program to write. Discovering the specification is part of the puzzle. This is quite unlike a homework assignment where a student might be told to “implement a sorting function”. Instead, the player is presented with a set of input-output test cases (Figure 5.1) and has to start guessing what the required program might be. Puzzles are grouped in Code Hunt in sectors and zones. Creating Code Hunt puzzles is a skilled task, at the level of setting a good homework assignment [58].

At each level of the game, the player starts with an empty method named “Puzzle” with the proper signature (for example, `int Puzzle(int x, int y)` for a secret function that

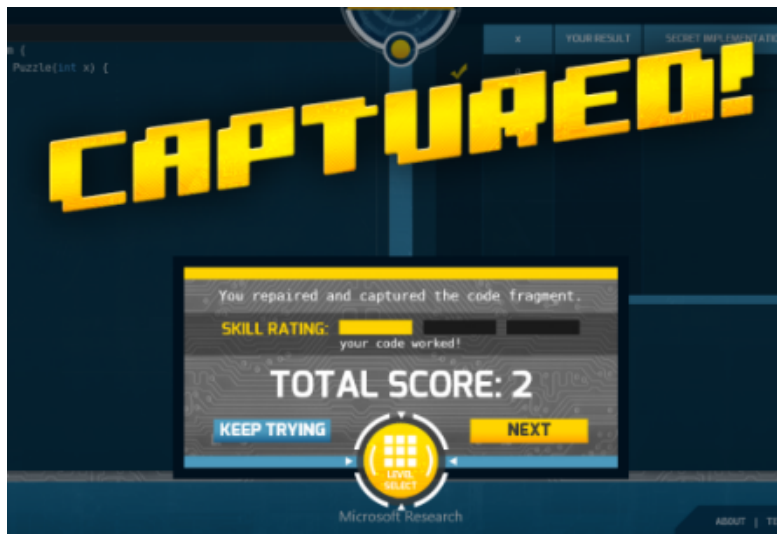


Figure 5.2: The Code Hunt “Captured!” screen.

takes two integers and returns an integer). At any time, the player can click the “Capture Code” button. The result of a capture may be

1. a “Captured!” message indicating that the program satisfies the secret specification (Figure 5.2),
2. a compiler error, as would be given in a normal programming environment,
3. or a set of test cases showing inputs on which the player’s program agrees and disagrees with the secret specification along with the corresponding outputs of both the player’s program and the secret specification (Figure 5.1).

The last case is the core of the game. Given those test cases, the player then attempts to intuit the pattern to determine the specification and produces a new program according to their theory. This process repeats until the player has both correctly guessed the specification and properly implemented a program for that specification, at which point the player has won the level and is encouraged to move on to the next level.

Kind of feedback	Example	Level-specific info
Compiler errors	No implicit conversion from “int” to “string”.	none
Counterexamples	Correct output for x=1 is 2, your code returns 0.	a solution
Skill rating	Correct! Skill rating: 1 out of 3.	many solutions
Line hints	Look at line 4 to capture the code.	many solutions
Recommendation hints	You may find a loop useful on this level. The expression <code><int> + <int></code> is rarely used to solve this level.	many attempt sequences

Table 5.1: Different kinds of feedback in Code Hunt

In its basic form, Code Hunt does not have feedback in the form of hints that assist a player when blocked. However, it does have recognition of how succinct the code was for the solution. This is shown in Figure 5.2: the skill rating for the solution is one, two or three bars, and for this puzzle, the player achieved only one. The score is the rating multiplied by the difficulty of the puzzle. As the game goes on, the multiplier gets greater. The player can replay the level to achieve a better score. In contests based on Code Hunt, players frequently return to a puzzle to achieve a maximum score. In the general case, a survey taken of 850 players indicates that 77.8% claim to keep tidying up their code.

Levels and data Code Hunt has a default level progression targeting the AP computer science curriculum (approximately equivalent to a first semester college computer science course), with 130 graded problems. Over the course of a year, from March 2014 to March 2015, it has been played by over 130,000 users [4]. They have produced 640,000 final correct solutions to the problems. For each solution, there can be anything from 2 to 100 attempts.

5.2 The Feedback System

5.2.1 Layers of Feedback

Code Hunt with our hint generation system offers five kinds of feedback listed in Table 5.1, which are specialized to the level by mining increasing amounts of data:

1. Compiler errors: No specialization to the level, so no data needed. If there is a compiler error, no other feedback is given as there is no program to analyze.
2. Counterexamples: Requires a single solution to the level in order to generate counterexamples with Pex [57]. The counterexamples are the core of the Code Hunt game experience, although they also act as hints and are used as such in other systems.
3. Skill rating for solutions: A value 1 through 3 that encourages smaller compiled code size. The maximum rating (3) is given for solutions that are within a small factor of the size of the smallest known solution.
4. Line hints: Tell the user the location(s) of one possible set of fixes to get a correct solution, similar to Autograder [51]. Requires multiple solutions (mined from other users) as it is better at directing users toward solutions similar to ones in the data.
5. Recommendation hints: Specify structures or expressions which the user might find useful or which are in the user's program but shouldn't appear in a solution. Requires multiple solutions and attempt sequences leading up to them in order to have enough evidence that a given structure or expression is usually useful or usually not useful.

The first three were already present in the Code Hunt game. This chapter focuses on the feedback we added, the last two, which together we call Code Hunt's hint generation system.

5.2.2 *Hint generation*

In any game, a major design goal is to keep the player engaged and in flow. Staying in flow is dependent on maintaining the proper level of difficulty: too easy and the game feels unfulfilling and boring, too hard and the game feels frustrating, but just right and the game feels challenging but doable and fun. In Code Hunt, the puzzle aspect of figuring out what specification the counterexamples are leading toward keeps the player engaged, but a player who is unable to figure out what to do may get frustrated and give up. The hints give additional feedback to help prevent the player getting stuck without giving away the answers, so the game remains challenging.

The following sections describe the two kinds of hints we generate; Section 5.3 describes how we generate these hints.

Line hints

For line hints, the problem we target is to guide players who are near a solution to focus on the parts of their program that need to be changed instead of the majority of their program which is already correct. Specifically, the feedback given will be one or more lines of their program that can be modified to produce a correct solution. This both lets players know they are on the right track and prevents them from getting distracted away from the correct solution.

The non-trivial part of this problem comes from the fact that there are many solutions to each level, and we want to produce the smallest change to any solution. While many solutions may be nearly identical—for instance, differing only by variable names—other solutions may actually be fundamentally different approaches to the problem that the creator of the level never thought of. We want to allow players creativity and not push them toward a hard-coded approved solution. Additionally, a student one typo away from a correct answer may be nowhere near a solution the creator of the level had ever thought of.

Section 5.3.1 describes how we apply TDS from `chaptftds` to solving this problem.

Recommendation hints

When a player is far away from a solution, it is not helpful to tell that player to change every line of their attempt. Instead, we need to devise other forms of hints that will be more useful to that player.

Recommendation hints are built around recommending for or against the use of specific features in a program. Features may be structures like loops or nested loops or expressions like `int.MaxValue` or `3 + <int>`.

If a player is off-track, a hint recommending against a feature that will not lead to a solution may nudge that player away from a bad strategy. We need to be careful to avoid recommending against features that merely lead to an unknown or rarely used solution: we do not want to tell players they are wrong when they are merely being original. To do so, we only recommend against a feature if many other players have used it in their early attempts but not their solutions.

In the other direction, if there are no features to recommend against, then we can instead nudge the player toward a solution by recommending features that are commonly used in solutions, preferring ones that are not used by players that never reach a solution.

Both kinds of recommendation hints require data mining the attempt sequences of many players on the same level. For each of those attempts, the system needs to analyze them to detect which features they use and then collect summary statistics across all of the players for use in deciding which hint to give.

5.3 Algorithm

The hint generation algorithm consists of three parts: Section 5.3.1 describes how line hints work, Section 5.3.2 describes how recommendation hints works, and Section 5.3.3 describes how the two are put together into a single hint system.

5.3.1 *Line hints*

The line hint generation algorithm is based off the test-driven synthesis algorithm described in Chapter 4, specifically DBS, which is described in Section 4.4. Two novel features of TDS make it appropriate for this use:

1. TDS takes an iterative approach to synthesizing programs, where at each intermediate step a program is generated that satisfies some subset of the test cases. We take advantage of this design to insert the player’s attempt as a program for the algorithm to build upon.
2. Unlike other program synthesis technologies that work across multiple domains, TDS does not rely on an SMT solver or similar technology [52, 60]. This allows for the flexibility to work in any domain without worrying about support for that domain from the underlying solver.

Additionally, the support for synthesizing loops and conditionals makes it flexible in the control flow structures of programs it can support. Just a few loop synthesis strategies can cover many of the loops that appear in simple programming assignments.

The rest of this section addresses how we adapt TDS to this setting.

Initial program

In TDS as used for end-user programming-by-example, the initial program P_0 is \perp , the empty program that fails on all inputs. For hint generation, the initial program is instead the player’s attempt.

Test cases

The synthesis algorithm depends on test cases to determine the correct program. It could get them from the test cases shown to the player, but the hint should direct the player toward

the actual correct solution, so those test cases may be insufficient. They could be augmented by querying Code Hunt just like what happens when the player clicks the “Capture Code” button, but, in practice, this is much too slow as generating counterexamples takes several seconds, so it cannot appear in the inner loop of the hint generator which has to be able to display a hint to the player within at most 30 seconds. In practice, all test cases that have been shown to all players as counterexamples are recorded. Using more test cases slows down the synthesis algorithm (in the worst case, the time to test a possible solution is proportional to the number of test cases to run it on), so initially only the 10 test cases most commonly shown to players are used, but the rest are available to verify any solution generated.

Datamining solutions

Component-based synthesis requires a set of expressions as an input which guides the search by limiting the set of programs to search through to those using those expressions. Limiting the search space is important because the search space of all programs using all constructs appearing in all Code Hunt levels is too large to search through quickly. Therefore, for each level, we mine player solutions for what expressions they used. As long as there are at least 50 solutions, expressions that appear in at least 5 other players’ solutions for a given level are considered by the hint generator. The cut-off is due to the observation that there is a large long-tail of useless expressions in the set of all expressions appearing any player’s solution.

Given that each level will have multiple solution strategies involving different expressions, it would be great if we could mine expressions only from solutions that used the same strategy that the player is currently attempting. While there’s no obvious way to automatically partition the solution strategies, one easy approximation is to consider the set of temporary variable types in the program. Different strategies might involve different variable types and working with different variable types will definitely involve different expressions, so it’s a reasonable way to partition the expression sets. Specifically, we collect an expression set for each level and each set (ignoring repeats) of temporary variable types that appeared in an attempt leading to a solution where that expression was used.

This is a relatively shallow mining of the data collected by Code Hunt, but it is sufficient to produce the expression set needed to run component-based synthesis efficiently without per-level human effort. A deeper analysis might look at what expressions are used together or at what changes players made to their own programs on the path to their solutions.

Selecting the best hint

Although the process has been described as generating a single hint, in reality, if there is one small change to correct the program, the synthesizer often finds several more soon after. Therefore, the synthesizer is not stopped immediately after finding the first solution. This gives an opportunity to rank the hints and select the best one to show to the player. As our goal was to present the smallest change to the player, we rank the hints by number of lines changed and then textual edit distance: changing a few characters on one line is better than rewriting an expression on another, even though the distance measured in edits to the AST may be the same or even larger, but both are better than changing two characters on two different lines.

5.3.2 Recommendation hints

Recommendation hints suggest a player use or not use a given feature in their program. By “feature”, we mean structures like “nested loops” as well as expressions (which may include typed holes) like “`<char>.toString()`”.

Hints can reference the following four structures: `if` statements, loops, nested loops, and recursion, where “loops” is defined such that it includes calls to higher-order functions internally implemented with a loop.

We want to advise against features that will not lead to a solution without discouraging different solutions and recommend features that do lead to a solution without suggesting features that appear in both good and bad attempts.

We discover these hints by data mining attempt sequences. For each feature, we want to count how many players found it useful and how many did not find it useful. Then we

can recommend features that were mostly useful and recommend against features that were mostly not useful.

Basic model

We want to build a statistical model which, for each feature, estimates the probability that a program containing that feature is leading to a solution (for positive recommendations) or a dead-end (for negative recommendations).

The naive computation would be to count all of the users who used the feature in a solution c^+ and divide by the number of users that used it ever n to get the proportion $\frac{c^+}{n}$ of players using that feature that used it in a solution. That gives a simple estimate of the probability p^+ that a player using that feature will find a solution using it. We can similarly estimate p^- , the probability that the player will not reach a solution using that feature, using $c^- = n - c^+$. We generate a hint only if the probability estimate $p > 0.75$ (where p stands in for either p^+ or p^-).

Our model is that a player aware of a feature has some probability of reaching a solution using that feature. We can guess what features a player is aware of by looking at what features appear in the attempts that player has submitted, and we can furthermore guess that those features are the same ones that player will use on their future attempts. Therefore, if a player is currently using a feature that has a low probability of reaching a solution, then a hint saying to not use it should increase that player's chance of reaching a solution. Similarly, telling a player about a feature which has a high probability of reaching a solution should also increase that player's chance of reaching a solution.

Limit to concise solutions

One problem with this is that “appears in a solution” is actually too weak a condition: due to the way the game works, a player may have code left over from an unrelated approach in their solution. While simply having a lot of data will hopefully filter this out, we can apply some domain knowledge as well: the less code there is in a solution, the more likely that

code is an essential part of the solution. We define “concise solutions” as solutions whose compiled code size is a small factor of the smallest known size for that level.

In order to define useful and non-useful, we partition attempts into three categories:

1. Concise solutions (compiled code size is a small factor of the best known for the level)
2. Other solutions
3. Incorrect attempts, which may or may not be followed by a solution

A player who uses a feature in a concise solution is considered to have found that feature useful, while a player who uses a feature only in incorrect attempts is considered to have found that feature not useful. The middle case is an attempt to filter out solutions that have unused features left over from the process of discovering the solution. More concise solutions are less likely to have unnecessary code in them.

This does not change the definition of c^- , $c^{(3)}$ is defined the same as c^- is defined above, but we do want to use $c_*^+ = c^{(1)}$ instead of $c^+ = c^{(1)} + c^{(2)}$.

Due to the structure of the game, there is no way to be sure players in (3) who have not found a solution yet will not soon find a solution and use the feature in it. Such a player may be off-track or may just be in the middle of playing. Additional assumptions about time since the last attempt might help there (e.g., if the last attempt was at least a day ago, then assume the player has given up on the level), but the current system does not make such guesses about the timing of the attempts.

Filtering out noise

In a large corpus, we want to be able to ignore just a few uses of a feature as noise: an expression used nowhere but a single solution might be great but is more likely just a fluke. Similarly, even a bad feature will end up in some solutions if only due to its presence in dead code in those solutions, even with the care taken above to limit that case.

To clean up issues relating to features with too little data to draw conclusions, we add noise to the counts. Specifically, we pretend that there are an additional 20 players who used the feature that are not in our data and that half of them used the feature in a solution. The selection of 20 is empirically based on seeming to give good results for our data and does not need to scale with the actual number of players that played the level because if there are a lot of players on a level, then that means a trend in the data is much less likely to be by chance.

In math notation, instead of c^- , c_*^+ , and n , we use $\tilde{c}^- = c^- + 10$, $\tilde{c}_*^+ = c_*^+ + 10$, and $\tilde{n} = n + 20$, respectively.

Confidence-based threshold

Additionally, we do not use a flat 75% threshold. Instead we adjust the probability estimate with its standard deviation $\sigma = \sqrt{\frac{p(1-p)}{n}}$ to acknowledge that we have lower confidence in an estimate based on fewer players.

Putting it all together, from the counts c_*^+ , and c^- of the players that used the feature in a good solution and those that did not use it in any solution, we define \tilde{p}_*^+ and \tilde{p}^- as $\tilde{p} = \frac{\tilde{c}}{\tilde{n}} = \frac{c+10}{n+20}$. From that we compute $\sigma = \sqrt{\frac{\tilde{p}(1-\tilde{p})}{\tilde{n}}}$. Then instead of the original check of $p > 0.75$, we instead check if $\tilde{p} - z\sigma > 0.75$ where we use $z = 1$ due to it empirically producing good results.

Using the math

The above decides for each feature whether it should be recommended for or against, but there are many features and we want to give only one hint. To do so, we first look at all of the features in the submitted program and for each of them compute $\tilde{p}^- - z\sigma$, the estimated probability that that feature will not lead to a solution. If any of those probabilities exceeds the cut-off of 0.75, then feature with the highest probability is chosen for the hint.

Otherwise, we go on to look at $\tilde{p}_*^+ - z\sigma$ for all features not in the submitted program and similarly select the highest of those if it is above 0.75.

If neither exceeds the 0.75 threshold, then no recommendation hint is generated.

As those values don't depend on the player attempt, they are cached so they do not need to be recomputed for every attempt. Instead, they are updated in batch by a separate process which handles adding all new attempts to the data.

5.3.3 Combining the hint mechanisms

While both hint mechanisms may generate a hint, at most one hint is shown to the user. The following kinds of hints are attempted in order, going to the next in the list only if there is no hint of the previous kinds:

1. A line hint that does not recommend changing all of the code.
2. A line hint on the only line of the program, specifying that the expression to be changed is a “constant”, “number”, “string”, “method call”, or “variable”.
3. A recommendation hint for an “unhelpful” feature that the player has used.
4. A recommendation hint for a “helpful” feature that the player has not used.

If there are no hints of those four kinds or it takes longer than 30 seconds to generate a hint, then no hint is shown.

5.4 Evaluation

We did not need to evaluate the ease of use of our choice of input because we our hint system does not require any additional input past what was already available in the system, which should also be available in any similar system. What we do evaluate is whether our system can generate hints and also whether these hints are actually any good.

While we hope our approach has broader applications, the context it was developed in is the Code Hunt programming game, which informs the kinds of hints we want to give and our metrics for evaluating our success. In particular, the goal of the game is to get users to

practice programming by solving puzzles whose solutions are programs. The goal is not to train the user to be better at solving Code Hunt puzzles, so we do not care if our hints make the puzzles too easy as long as the user continues to play the game and therefore practice writing code.

Additionally, we can measure how often our program synthesis algorithm is actually able to generate a solution despite not being given an error model.

Section 5.4.1 describes the A/B test of disabling hints for some players in order to collect data on the effect our hint mechanism has on player behavior. Section 5.4.2 details how many hints of each type were generated. Section 5.4.3 looks at the effect of hints on player engagement. Section 5.4.4 presents an experiment which lends some credence to our intuition that line hints get generated when “near” a solution and the system successfully falls back on recommendation hints when “far” from a solution.

5.4.1 A/B test

We wanted to know how hints affect players interacting with the game. A typical test scenario would be to watch many people where each one played the game with and without hints. This test scenario has two problems:

1. Bringing people in and watching them is expensive and time-consuming.
2. It is not meaningful to have the same people play the game twice (with and without hints) because they will already know the answers when playing the second time.

We can instead take advantage of the fact that user interactions with the game are logged, so our experiment can be done on normal users. Furthermore, because the hints are not a promised feature, the users do not even need to know they are part of an experiment, both making recruiting easier and not perturbing the data by letting users know they are being studied.

Design

To that end, we ran an A/B test on the live Code Hunt website¹ where for a two week period all new users were divided into one of three groups:

1. Hints always (whenever a hint is generated)
2. Hints never
3. Hints on some levels

We only collected data on new users in order to avoid users having already seen hints and therefore expecting them.

5.4.2 Results

Data was collected on 407 users who began playing between the start of the experiment and two days before the data collection period ended.

Those users submitted a total of 34,886 attempts. 7,734 (22%) of those attempts were correct solutions. Of the remaining 27,152 incorrect attempts, 5,143 (19%) of them did not compile, so semantic hints could not be generated, instead compiler errors were shown.

Of the 21,868 incorrect attempts that did compile, the program synthesis algorithm was able to solve 9,604 (44%) of them. As not all solutions lead to good line hints, only 6,284 (65%) of those were used to produce line hints. Similarly, 2,762 (13%) had recommendation hints for “helpful” features available and 13,580 (62%) had recommendation hints for “unhelpful” features available for a total of 16,342 (75%) of attempts having a recommendation hint available, while only 12,394 (75%) of those were selected as the hint generated.

The breakdown of hint kinds generated² for the 21,868 incorrect attempts that did compile were 1,498 (7%) normal line hints, 4,786 (22%) line hints referencing the expression kind to change, 2,273 (10%) recommendation hints for “helpful” features, and 10,121 (46%)

¹<https://www.codehunt.com>

²These numbers are about hints generated, not hints shown to the player. Hints are still generated for players with hints disabled, they just aren't shown to the player.

recommendation hints for “unhelpful” features. No hint was generated for the remaining 3,190 (15%) attempts.

5.4.3 *Player engagement*

What we actually want to know is whether hints affect player behavior, and, perhaps more importantly, is that effect positive. In order to answer that in the positive, we looked at the effect of hints on player engagement; that is, how long the players continue playing. In line with our hypothesis that hints reduce frustration, we expect some users who do not see hints may stop playing while users who do see hints are less likely to do so.

Figure 5.3 shows how long players kept playing the game. The y -axis is the proportion of players in each condition while the x -axis is the time between the first and last attempt submitted by a player. The dashed vertical line in the middle is the 1-day mark. Looking there, we see that about 40% of the players in the “sometimes hint” condition played for at least a day, as did about 20% of those in the “always hint” condition, but only about 10% of those in the “never hint” condition did. The lines going off the graph to the right indicates players that continued playing after the 48-hour period shown in the graph.

Looking at time played in a different way, Figure 5.4’s x -axis is the number of levels won in the same 48-hour period. The vertical dashed lines mark a few of the harder levels early on in the game where the sharp drop in the “never hint” line is visible but smaller in the other lines.³ Looking to the right side of the graph, we can clearly see that the players who received hints went on to beat more levels than those that did not receive hints. Of course, this makes sense, since the hints make the game easier.

From these two graphs, we can conclude that hints lead to players playing the game longer, which was the goal. Furthermore, giving hints sometimes instead of always seems to be somewhat better at getting players to continue playing, perhaps due to constant hints making the game too easy.

³Although the game does allow players to skip around to some extent, most play the levels in order and stop playing when they get stuck instead of switching to a different level.

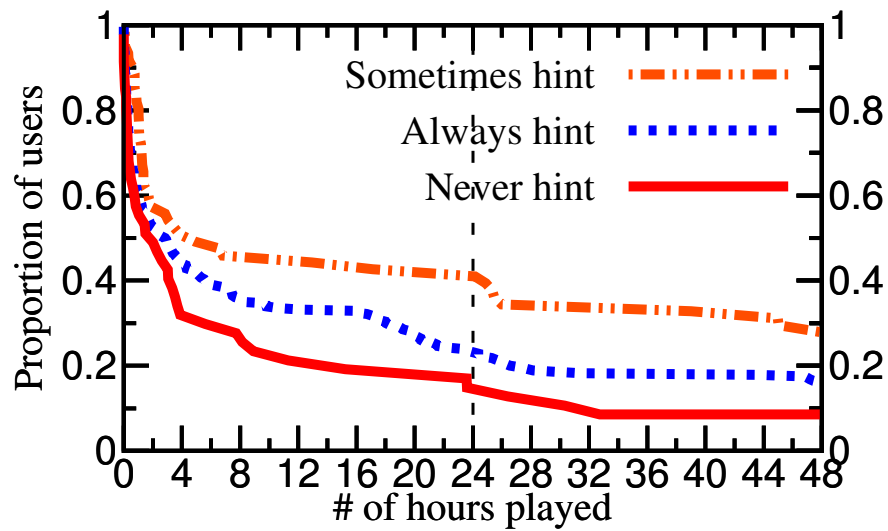


Figure 5.3: CDF of time between first and last play by A/B test condition

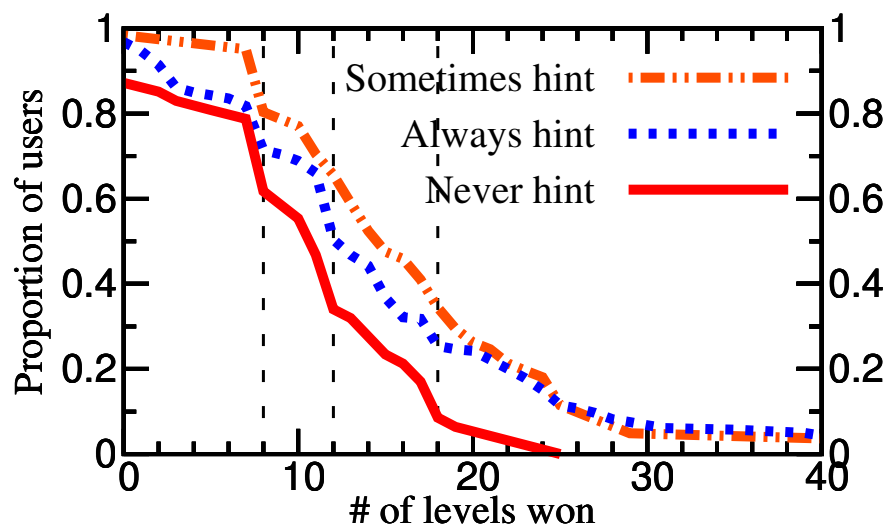


Figure 5.4: CDF of # of levels won by A/B test condition

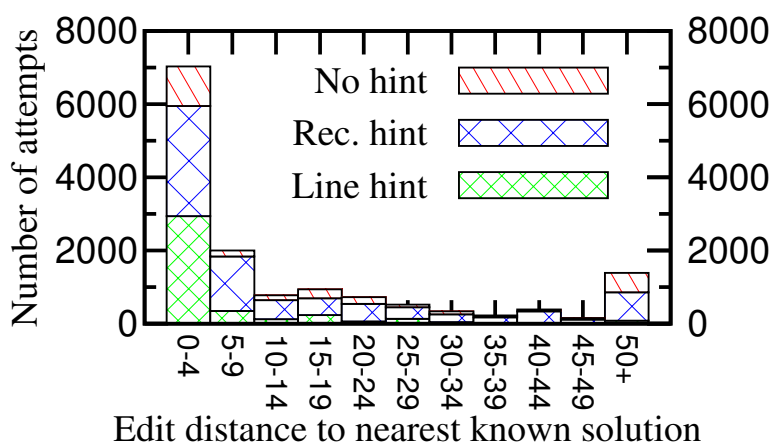


Figure 5.5: Number of attempts which got each hint kind by edit distance

5.4.4 Distance to solution

We hypothesize that line hints will be generated when near a solution but not when far away from a solution, which is part of why we also have recommendation hints. This is difficult to verify because “near” and “far away from” a solution are ill-defined.

Defining distance In order to have some support for this claim, we give an imperfect but concrete definition of distance to a solution using textual edit distance.⁴ We normalize programs by taking just the body of the method (which, among other things, gets rid of `import` statements) and removing all whitespace. Then for a given attempt, its distance is defined as the minimum edit distance to any known solution.

Results Figure 5.5 and Figure 5.6 show the rate of hint generation bucketed by that distance measurement. The former has the number of attempts on the y -axis while the latter is the same data with the y -axis instead being the proportion of attempts in that edit distance bucket. We do indeed see that for small edit distances many more line hints are generated while the overall proportion of hints generated only goes down slightly, showing

⁴https://en.wikipedia.org/wiki/Edit_distance

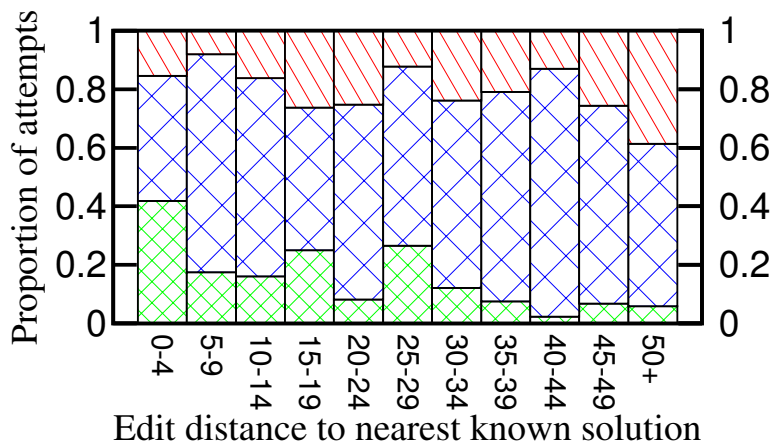


Figure 5.6: Proportion of attempts which got each hint kind by edit distance

that the recommendation hints are indeed covering for the difficulty of generating meaningful line hints.

5.5 Conclusion

We developed a hint system for the Code Hunt programming game based on the input constraint that it was not reasonable to either expect per-level advice to the hint nor to require the hint system to be specialized to any particular domain, other than assuming the programs would be small. Following these requirements, we used our TDS program synthesizer along with a statistics-based approach to generate hints for most attempts, and furthermore showed that the hints we generate increase player engagement.

Chapter 6

CONCLUSIONS AND FUTURE WORK

We set out to show that our design methodology for program synthesis algorithms of focusing on the user input, and, in particular, looking for inputs that require a small amount of additional user effort does in fact lead to making the problem much easier for a computer. We demonstrated this methodology and showed it effective in the domains of API discovery, programming by example, and feedback generation for education. For all three tasks, we were able to develop simple algorithms using their respective new inputs that are fast and effective while avoiding expending effort or losing generality by specializing our algorithms to specific domains.

Future work

As noted in Chapter 3 and Chapter 4, while we believe the input forms we devised are easy for humans to provide, it would be best to run user studies to confirm these hypotheses.

Partial expressions actually have been used in a user study as the Code Hint [12] project uses partial expressions and ran a user study, but they were not a focus of the study. Code Hint's focus is in fact on dynamic code completion and thereby takes a different approach to considering what is easy for a programmer to provide by having them give dynamic constraints on the values the expression evaluates to instead of static constraints on its structure like partial expressions.

As mentioned in Chapter 5, our hint system uses only a little bit of the data available to be mined. The Codewebs [41] project suggests some ways that could be extended by looking for similarities between attempts and grouping them to get more information. One application of that may be figuring out a better way to determine which solution strategy a

user was going for and thereby only provide hints in line with that strategy. There is plenty more to be explored in that area.

Bibliography

- [1] <http://racket-lang.org>.
- [2] <http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html>.
- [3] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1976–1982. AAAI Press, 2013.
- [4] Judith Bishop, R Nigel Horspool, Tao Xie, and Jonathan de Halleux. Code hunt: Experience with coding contests at scale. *Proc. ICSE, JSEET*, 2015.
- [5] Christie Bolton. Typsy: a type-based search tool for Java programmers. In Cathy Miller, editor, *Selected 2001 SRC Summer Intern Projects*, Technical Note 2001-004. Compaq Systems Research Center, December 2001.
- [6] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.
- [7] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. *ICSE*, 2011.
- [8] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A search engine for Java using free-form queries. *ETAPS/FASE*, 2009.

- [9] A. Cypher, DC Halbert, D. Kurlander, H. Lieberman, D. Maulsby, BA Myers, and A. Turransky. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.
- [10] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. ECOOP, 2011.
- [11] G. Furnas, T. Landauer, L. Gomez, and S. Dumais. The vocabulary problem in human-system communication. *CACM*, 30, Nov 1987.
- [12] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.
- [13] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. PLDI, 1992.
- [14] Sumit Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.
- [15] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. POPL, 2011.
- [16] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. SYNASC, 2012.
- [17] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8), August 2012.
- [18] 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.

- [19] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *Computer Aided Verification (CAV) Tool Demo*, 2011.
- [20] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. PLDI, 2011.
- [21] Paul Hyman. In the year of disruptive education. *Communications of the ACM*, 55(12):20–22, 2012.
- [22] D. Janzen and H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9), sept. 2005.
- [23] Manu Jose and Rupak Majumdar. Bug-assist: assisting fault localization in ansi-c programs. In *Computer Aided Verification*, pages 504–509. Springer, 2011.
- [24] Susumu Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI*. 2004.
- [25] Susumu Katayama. Systematic search for lambda expressions. TFP, 2005.
- [26] Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI*. 2008.
- [27] Etienne Kneuss, Viktor Kuncak, Ivan Kuraj, and Philippe Suter. On integrating deductive synthesis and verification systems. *arXiv preprint arXiv:1304.5661*, 2013.
- [28] T. Lau et al. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, 2008.
- [29] T. Lau, S.A. Wolfman, P. Domingos, and D.S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1), 2003.

- [30] Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI*, volume 4, 2004.
- [31] 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*, pages 36–43. ACM, 2003.
- [32] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1), January 2012.
- [33] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [34] Greg Little and Robert C. Miller. Keyword programming in Java. ASE, 2007.
- [35] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. PLDI, 2005.
- [36] Robert Martin. The transformation priority premise. <http://blog.8thlight.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>, 2010.
- [37] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGPLAN Notices*, volume 22, pages 122–126. IEEE Computer Society Press, 1987.
- [38] Robert I McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O’Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.
- [39] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. ICML, 2013.

- [40] Tom M Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.
- [41] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International World Wide Web Conference (WWW 2014)*, 2014.
- [42] Robert O’Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. ICSE, 1997.
- [43] Roland Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1), March 1995.
- [44] Rina Panigrahy and Li Zhang. The mind grows circuits. *CoRR*, abs/1203.0088, 2012.
- [45] Michael Pardowitz, Bernhard Glaser, and Rüdiger Dillmann. Learning repetitive robot programs from demonstrations using version space algebra. *Hand*, 200:250, 2007.
- [46] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. PLDI. ACM, 2012.
- [47] Steven P. Reiss. Semantics-based code search. ICSE, 2009.
- [48] Mikael Rittri and Mikael Rittri. Retrieving library identifiers via equational matching of types. CADE, 1992.
- [49] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.
- [50] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’03, pages 76–85, New York, NY, USA, 2003. ACM.

- [51] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26. ACM, 2013.
- [52] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.
- [53] Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1887–1896. ACM, 2014.
- [54] Suresh Thummalapenta and Tao Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. ASE, 2007.
- [55] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Pex4Fun: Teaching and learning computer science via social gaming. CSEE&T, 2012.
- [56] Nikolai Tillmann, Judith Bishop, R Nigel Horspool, Daniel Perelman, and Tao Xie. Code hunt: Searching for secret code for fun. In *SBST*, 2014.
- [57] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .NET. TAP, 2008.
- [58] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Constructing coding duels in pex4fun and code hunt. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 445–448. ACM, 2014.
- [59] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

- [60] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152. ACM, 2013.
- [61] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 54. ACM, 2014.
- [62] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5), May 2010.
- [63] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. ICSE, 2009.
- [64] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
- [65] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. OOPSLA, 2011.
- [66] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, April 1995.
- [67] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6:333–369, 1996.