Program Synthesis for Programmers

Hila Peleg
Program Synthesis for Programmers

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Hila Peleg

Submitted to the Senate of the Technion — Israel Institute of Technology
Av 5779 Haifa August 2019
This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Conference/Editor</th>
<th>Pages</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hila Peleg</td>
<td>On the importance of common sense in program synthesis</td>
<td>Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018</td>
<td>page 7</td>
<td>2018</td>
</tr>
<tr>
<td>Hila Peleg, Shachar Itzhaky, and Sharon Shoham</td>
<td>Abstraction-based interaction model for synthesis</td>
<td>Verification, Model Checking, and Abstract Interpretation, pages 382–405, Cham, 2018</td>
<td>Springer International Publishing</td>
<td>2018</td>
</tr>
<tr>
<td>Hila Peleg, Dan Rasin, and Eran Yahav</td>
<td>Generating tests by example</td>
<td>Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings</td>
<td>pages 406–429, 2018</td>
<td>2018</td>
</tr>
<tr>
<td>Hila Peleg, Sharon Shoham, and Eran Yahav</td>
<td>Programming not only by example</td>
<td>Proceedings of the 40th International Conference on Software Engineering, pages 1114–1124</td>
<td>ACM, 2018</td>
<td>2018</td>
</tr>
</tbody>
</table>

**Acknowledgements**

I would like to thank my advisor, Prof. Eran Yahav, for guiding me when I needed it, and not guiding me when I needed that more. For listening, and for understanding. It was a privilege to work with him and learn from him.

I would also like to thank Dr. Sharon Shoham, my long-time coauthor and friend, for teaching by example how to write, how to prove, and how to get more done in a day than I would have thought possible.

I would like to thank Prof. Mooly Sagiv for encouragement and career advice, and for a desk in his lab. I would like to thank the indomitable Yifat Chen Solomon for all the paperwork she helped me with, and for all her help in setting up the user study at the heart of this thesis. I would also like to thank my many office-mates over the years: Asya Frumkin, Elazar Gershuni, Shachar Itzhaky, Omer Katz, Artem Khyzha, and Orr Tamir.

I would like to thank my friends: Kalev Alpernas and Yotam Feldman for many coffee breaks, emotional support and a lot of proofreading, Ayala Sela and Boaz Goldstein for being supportive from afar, and Gili Yavneh for doing both.

I would like to thank Willis Haviland Carrier for inventing the electrical air conditioning unit and thereby making life, complex thought, and research possible in the Tel Aviv heat.

Finally, I would like to thank my wonderful partner, Arnon Amikam, who has supported my every decision and every degree, and endured alongside me every deadline, internship, and rejection, and reminded me to celebrate every success; and our cats, who purr and put everything into perspective.
The research leading to the results presented in this thesis is partially supported by the European Union’s Seventh Framework Programme (FP7) under grant agreement no. 615688 (PRIME) and the Israel Science Foundation grant no.1319/16. The generous financial help of the Chil and Berta Weissmann Doctoral Fellowship Fund is gratefully acknowledged. The Technion’s funding of this research is hereby acknowledged.
# Contents

List of Figures

Abstract 1

1 Introduction 3
  1.1 Iterative Interactive Synthesis for Programmers . . . . . . . . . . . . . . 7
  1.2 Thesis Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2 Preliminaries 11

3 Generating Tests by Example 15
  3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  3.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  3.3 Property-Based Testing . . . . . . . . . . . . . . . . . . . . . . . . . . 22
  3.4 Compatible Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
    3.4.1 From Test Trace to Parameterized Test . . . . . . . . . . . . . . . 24
    3.4.2 Separation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  3.5 Abstracting the test data . . . . . . . . . . . . . . . . . . . . . . . . . 27
    3.5.1 Handling Impreciseness . . . . . . . . . . . . . . . . . . . . . . . 29
  3.6 Sampling from the Abstraction . . . . . . . . . . . . . . . . . . . . . . 29
  3.7 The role of the user . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    3.7.1 The role of the user in creating regression tests . . . . . . . . . . 32
  3.8 Experimental evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 32
    3.8.1 Examining Apache test suites . . . . . . . . . . . . . . . . . . . . 32
  3.9 Instruction and value Coverage . . . . . . . . . . . . . . . . . . . . . . 33
  3.10 Discovering Bugs: A Case Study . . . . . . . . . . . . . . . . . . . . . 36
    3.10.1 MATH-1256: Interval bounds . . . . . . . . . . . . . . . . . . . . 37
    3.10.2 MATH-785: Discovering a deep bug with manual intervention . . 38
  3.11 Conclusions from JARVIS . . . . . . . . . . . . . . . . . . . . . . . . . 38

4 Programming Not Only by Example: The Granular Interaction Model 41
  4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
  4.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5 A Formal Model for Interactive Synthesis

5.1 Introduction
5.1.1 Our approach
5.1.2 Main contributions

5.2 Foundations for Synthesis with Abstract Domains
5.2.1 An abstract domain for programs
5.2.2 Examples
5.2.3 Computability of the model

5.3 An Abstract Model of Interactive Synthesis
5.3.1 Progress-making sessions

5.4 Termination
5.4.1 WQO predicates
5.4.2 Locally strongest user

5.5 Successful Convergence and Backtracking
5.5.1 Unbounded Backtracking

6 Recommendations for Future Synthesizers
6.1 Progress models
6.2 Realizability gap
6.3 Sharing more with the user

7 Related Work

8 Conclusion

Hebrew Abstract
List of Figures

3.1 Several unit tests from the test suite of the Apache commons-math project, using the JUnit testing framework. .......................... 19
3.2 The ScalaCheck properties synthesized from the test traces shown in Fig. 3.1. .......................................................... 23
3.3 Unit test code for the Interval class from the Apache Commons-Math project .................................................. 36
3.4 The ScalaCheck generator and property generated by JARVIS from the unit tests in Fig. 3.3. .............................. 37
3.5 (a) The ScalaCheck generator and property generated by JARVIS for FDistribution. (b) The code from (a) after manual modifications to the generator in lines 2 and 3. ............................... 39
4.1 Program with debug information, a sequence selected for removal . . . . 52
4.2 Median iteration time per task in each test group. Significant change from PBE is indicated by *. ................................. 58
4.3 Examples used (med, min, max) by GIM users (all operations). None used 100% examples. ................................. 59
4.4 No. users that reached the target program or an equivalent (non-functional differences) result. ............................ 59
Abstract

Recent years have seen great progress in automated synthesis techniques that can automatically generate code based on some intent expressed by the user, but communicating this intent remains a major challenge. When the expressed intent is coarse-grained (for example, restriction on the expected type of an expression), the synthesizer often produces a long list of results for the user to choose from, shifting the heavy-lifting to the user. An alternative approach is programming by example (PBE), where the user leverages examples to interactively and iteratively refine the intent.

Existing program synthesis tools are usually designed around the synthesizer and its internals. However, these are tools intended for users, who are the ones who must specify (and respecify) the specifications. Synthesis tools are often designed either with no particular group of users in mind, or with the purpose of generating code for users who cannot write and read it.

This thesis focuses instead on designing synthesis tools specifically for programmers. This allows making assumptions on both the input the user can generate and the output they can consume. Concepts that are part of the programmer’s life such as code review and unit tests can be leveraged.

We begin by discussing the human common sense of making generalizations can be aided by the user, as opposed to machine generalizations that rely on inductive bias [Pel18]. We draw from this the conclusion that some leaps of generalization or induction cannot be made by the synthesizer and are best left to a human. We demonstrate this on a non-iterative synthesizer JARVIS [PRY18], which employs multiple biases but still required manual intervention by its user to improve the final result.

As a result of this, we set out to treat synthesis both as an iterative and an interactive process, involving the programmer in a gradual refinement of the specifications. But this approach also brings with it restrictions for the synthesizer, which pose new design challenges: examples, a common tool, are not expressive enough for programmers, who can observe the generated program and refine the intent by directly relating to parts of the generated program. Additionally, can the users correctly judge when the program is correct? We suggested a new Granular Interaction Model (GIM) [PSY18] and performed a controlled user study to assess its effectiveness.

In addition, we modeled the interaction of the user with a synthesizer [PIS18], formalizing the refinement of specification by the user and the respective reduction
of the candidate program space. This model allowed us to present two conditions for
termination of a synthesis session, one hinging only on the properties of the available
partial specifications, and the other also on the behavior of the user. Finally, we showed
conditions for realizability of the user’s intent, and limitations of backtracking when it
is apparent a session will fail.
Chapter 1

Introduction

Program synthesis is the problem of computing a program that implements a given specification. The classic synthesis problem [Chu57] searches for an implementation to a full specification, usually expressed in some logic [LNP+12, VYY10, BGL+98, IGIS10, Pa90, CGT+16, ISSL+16a, VYB06, VYBR07, HP11]. Since full specifications are cumbersome and error prone, new formulations of the problem have evolved, using a partial specification that is easier for the user to provide.

Synthesizers are usually given specifications in one or more of these partial specification formats:

**Sketching** A sketch is a partial program where holes are left to be completed by the synthesizer. The term was coined by Sketch [SLTB+06] but is now used for all manner of ways to provide a skeleton of the resulting program [SGF10, SA16, SGF13, HK17, WSMK18]. Sketching is a form of partial specification in that it allows the user to direct the synthesizer to find a program that fits the provided skeleton. It is accompanied by other forms of specification to dictate how the sketch should be completed.

**Partial logical specification** Assertions of properties on intermediate states [SGF10, ČHR+14] or on holes in a sketch [SL08] are logical specification that specify only a small portion of the behavior of the program. The partiality of these assertions is generally supported by the limitations of the sketch they are provided for or limitations on the language the synthesizer can use to create a candidate in synthesizers employing Syntax-Guided Synthesis (SyGuS) [ABJ+15].

**Types** When attempting to synthesize code using in a strongly-typed language, especially an object-oriented one, types can be an expression of intent: the user would like a function accepting (i.e., with a scope including) parameters of set types, and returning a value of another type. Synthesizers such as [FMW+17a, GRB+14] attempt to fulfil such partial specifications. Another approach uses refinement types, types that are narrowed with logical predicates, which specify the data dependencies of the synthesized execution.
Programming by Example (PBE) In recent years, examples have become one of the major solutions in communicating intent [WCB17, OZ15, FCD15, Gul16, Gul11, Gul12, PG15, YWD18, URD+13, AGK13]. This is a particularly good fit for works that target end-users (e.g., users of Microsoft Excel) and simple data-wrangling tasks, but has also become a popular form for expressing intent in works aimed at power-users and programmers [FMW+]17a, FMW+17b].

This makes sense from the perspective of the user: programmers use examples to understand the problem, to explain complex parts of the solution to each other, and to test their code (e.g., unit tests are specific examples, integration tests generally run on data samples). However, human users are aided by two things which are often missing when a synthesizer approaches a problem: (1) a natural-language description of the problem, and (2) the ability to make generalizations of behavior from few examples. While humans are generally capable of performing small-scale generalizations without (1), the only way a synthesizer can cope with (2) is via bias.

Generalization and Inductive Bias Inductive bias is the set of assumptions made by the learner in the course of generalization [Mit80]. Specifically, an unbiased language of classifiers is one that can describe any set of concrete examples precisely. In a learning algorithm, bias in the language of available classifiers is a positive thing: if the selected classifier describes precisely the examples seen, that means the classifier will never make a decision about any example it hasn’t seen. For example, if the language of a synthesizer allows constructing a switch statement which returns the desired output for each input, that means that rather than generalizing the intended behavior, the synthesizer can simply return a program that only handles the examples provided.

Synthesizers are generally biased towards simpler programs (i.e., a switch is a long program, there may be a shorter program generalizing the demonstrated behavior) but there are other forms of bias that are desirable when attempting to generalize from examples: assumptions that limit the input/output domain direct and simplify the generalization, and assumptions on how the resulting model will be used allow specific behaviors (e.g., false positives, or offering as candidates solutions that are incorrect, are better than false negatives, or ruling out the correct solution, as a human will review the result).

Bias in Synthesis Even when program synthesis is not implemented via machine learning algorithms, it is still affected by inductive bias. Two types of inductive bias form the collective bias of any given synthesizer:

1. Representation bias: The candidate program space, or search space, is the space of programs available for the synthesizer to enumerate or search. The search space creates the same bias a language of classifiers creates in a learning algorithm, one
of representation: we may not be able to represent a set of examples precisely, which means the resulting program will represent a wider set or behavior (i.e., perform a generalization).

2. **Learning bias:** The bias of the *search algorithm*, is the order and mechanism in which the candidate program space is traversed. This form of bias is essentially one of ranking: out of the many programs that generalize the specification in some level, the search algorithm will determine which one is selected as the result of the search. This does not have to include an actual ranking function, but can rather be a product of how the space is explored: in an enumerating synthesizer, for example, smaller programs are often considered first, which means overfitted simple programs that satisfy some ambiguous component of the example set often “win” over a more complex program that satisfies the intended behavior. Synthesizers such as [Gul11] have attempted to modify this bias by encoding it specifically: e.g., FlashFill favors programs slicing substrings of the input over programs with constants constants when synthesizing a string transformation.

These two forms of bias are not independent. Their combination creates additional bias: the search space can be seen as partitioned into classes of functionally equivalent programs, in which a certain program may “mask” others by being favored by the search algorithm, causing a desired program to become unreachable in the search. Chapter 4 shows an example of this, and how it makes certain modes of intent communication insufficient for the user.

**An attempted solution: Multiple Biases** One of the common solutions for overfitting in machine learning is learning several classifiers, each under a separate bias, and comparing them. A recent work has done this for SyGuS synthesizers by searching for a solution over multiple sub-grammars of the initial grammar [PMNS19], where the first program to be found is returned without waiting to see if there are others under other grammars.

JARVIS, one of the tools presented in this thesis (Chapter 3), was an earlier attempt to solve this problem. JARVIS is a tool that expands value coverage of a unit under test by generalizing repetative unit tests into a property-based test, a technique combining parametric tests with value generators, to create an efficient and maintainable way to test general specifications. The generalization of the concrete unit tests in JARVIS was not done in a single search space of abstractions, but rather with multiple languages (*abstraction templates*) that are differently biased.

One of the challenges JARVIS faced is what to do when more than one set of biases succeeded in abstracting the example set. When only one such abstraction succeeded, the output program is clear. But any ranking between the different abstractions was domain specific. It required human intervention at the tool level—creating the ranking.
But introducing a new set of benchmarks in another domain usually caused the ranking to fail and required re-ranking the abstractions to fit each new domain.

The fact that JARVIS performed its abstraction on very few examples meant it occasionally required manual intervention in correcting the output code, as shown in Section 3.10. A possible solution for the competing biases and domain specific ranks could come from a different source: applying human “common sense” to mitigate an unbiased space.

We wish to explore methods in which the human user can apply common sense: to select one of several possible biases, to refine specifications so that existing biases are no longer relevant, or to simply intervene manually in the result. In addition, we ask what the limitation of this human intervention is, since common sense can depend on many factors, first and foremost an understanding of the problem being solved and the solution being proposed by the synthesizer.

**Involving the programmer** Modern formulations of the program synthesis problem generally present themselves as one-shot solutions that can find the best program from one small set of examples. FlashMeta [PG15, LG14, Gul11] synthesizers are evaluated on as few as one or two examples that lead to the target program, and even the most complex synthesis systems [FMW+17a, SA16] describe their synthesis process as a single execution over user specifications.

However, it is important to notice that interactivity is always inherent in the workflow of a real user with a synthesizer: if the solution returned by the synthesizer is not satisfactory to the user, they will change the specifications and re-run the synthesis process, and be presented with a new answer candidate. This iterative process of candidate solution and refinement is rarely discussed, as theoretical focus in state of the art research is focused on each single attempt to reach the user’s intended program with as partial a specification as possible, via rankings and biases.

**Inversion of control** A different model of synthesizers has been named by Gulwani [Gul12] synthesizer-driven synthesis. As opposed to user-driven synthesis, in which synthesis simply takes a specification (or re-specification) from the user and returns a candidate, the synthesizer-driven model is very similar to active learning [Set09]. In active learning, the learner—in our case, the synthesizer—selects samples that should advance the quality of the model and asks a “teacher” to classify them. This has been applied to many fields, such as abstraction [PSY16], and in certain formulations of the problem is also a natural fit for synthesis. If a logical specification of the desired solution exists, a solver can be used as the teacher and find an example for which the expected output is known and is different from the behavior of the program. This is the modus operandi behind counterexample-driven synthesis [SLTB+06]. Likewise, Vu et al. [LPP+17] define a model of interactive synthesis where the user serves as the teacher.

If we choose to view the user-driven model as interactive as well, i.e., a loop that
iterates until the user is satisfied with the result, we can strive to improve the interactivity. While the user-driven model inherently leaves the control firmly in the hands of the user, the synthesizer can still aid the user in getting the best result. This requires us to examine the interactive, user-driven session, and design the best feedback tools for users of such a session.

1.1 Iterative Interactive Synthesis for Programmers

In this section, we define the components of an iterative, interactive synthesis session.

A synthesis session We view the iterative synthesis algorithm as an iterative refinement (i.e., adding of predicates) of the specification in each iteration of the process. This creates a synthesizer state from which the next candidate displayed to the user is selected. When synthesis is defined as an interactive session, notions of convergence, both at the practical level (i.e., the session ends when the user accepts the candidate program) and the theoretical level are also needed alongside it.

We show several properties of interactive synthesis. We define the point from which a synthesis session can no longer converge, even if the user has, from their point of view, only provided correct specifications, and properties of the point we must backtrack to when that happens. We offer two separate sets of limitations on the model that lead to convergence (i.e., a finite session) in every session. These properties allow us to better understand the “long-tail” of synthesis sessions, those that, despite the best engineering efforts of synthesizer designers, do not end within one or two iterations.

Strengthening User Feedback Existing methods for the user to provide specifications—examples, types, and logical specifications—are holistic forms of specifications. They are end-to-end, “take it or leave it” specifications of the program. This means that when presented with a program, the user can only completely accept or completely reject it, and any subprograms are opaque to the user. This is a sound method of specification for end users. However, when one can assume that the user is a programmer, one can assume a set of skills that allow the user to think of the program not only in holistic, functional terms.

We create a new, granular interaction model (GIM) for synthesis. GIM allows for a richer form of feedback in both directions: from the synthesizer to the programmer and back.

From the synthesizer to the programmer: A candidate program will be presented together with debug information, showing execution values at different program points. This helps the programmer understand whether the candidate program does the expected computation at intermediate states, instead of relying only on its final output.

From the programmer to the synthesizer: A programmer can provide: (i) input-output examples (as in PBE), and (ii) granular feedback on the candidate program by explicitly
accepting/rejecting parts of its code.

The motivation for this form of interaction is the observation that synthesis with input-output examples requires the user to provide examples that differentiate the candidate program from the desired program. Our underlying assumption is that, in some cases, it is easier for a programmer to explicitly indicate what is good or bad in a candidate program, instead of implicitly trying to express this information through input-output examples.

This assumption is strongly supported by a controlled user-study with 32 developers from both academia and industry. To conduct this study, we developed a synthesizer that interacts with the user in three different ways: holistic (PBE), granular, or both. Our synthesizer also measures interaction times, and records the user-interaction so we can later analyze it.

**An abstract domain of general predicates** The predicates developed for GIM are only an example of feedback mechanisms that can be made available to the user. We formalize the notion of general predicates: allowing any (quickly) decidable unary predicate on programs as a specification mechanism for the user. Further examples include data predicates on subexpressions of the program, intermediate type constraints, and, of course, input-output examples when expressed as a predicate.

This generalization allows us to express the specifications and the state of the synthesizer as an abstract domain of predicates, with a concrete domain of programs in the synthesizer’s language. Given a domain of programs \( M \) and a domain of predicates on programs \( P \), we define the concrete domain of the synthesis algorithm to be sets of programs \( (2^M, \subseteq) \) and the abstract domain to be sets of predicates \( (2^P, \supseteq) \), with an abstraction function that produces the incomputable set of all predicates that hold for the set of programs, and a concretization function that produces the equally incomputable set of all programs that satisfy a conjunction of all predicates (in propositional logic) in a given set. Since both these sets are likely not computable, a real synthesizer relies on the synthesizer’s representation of the state to replace a concretization, and the user to replace the abstraction.

This generalization allows us not only to prove the properties stated above, but to understand the limitations of predicates we wish to add to the synthesizer.

**Recommendations for future synthesizers** Finally, we discuss the implications of the results presented in this thesis, both theoretical and experimental, for the designers of future synthesis tools. We believe our results indicate many possible improvements to future synthesizers.

### 1.2 Thesis Contributions

The contributions of this thesis are as follows:
• In Chapter 3 we describe JARVIS, a framework for abstracting repetitive unit tests and synthesizing property-based tests based on that abstraction. This includes: (1) An inclusion relation between parameterized tests that allows the sharing of examples between different abstracted generators without hindering the ability to abstract. (2) A technique that generalizes test values from individual unit tests into value generators for a property-based tests using safe generalization (separating positive and negative examples). (3) The tool, JARVIS, that synthesizes parametric tests, oracles and abstraction-based generators from unit tests, while preserving the subtle cases that are captured in these tests. In the experimental evaluation of JARVIS, it needed manual intervention to improve the result on one occasion, and manual intervention was described as one of its modes of conflict resolution. As described above, this drove developing a new model in which the programmer is more involved, rather than intervening at the end of the process.

• In Chapter 4 we define and implement the granular interaction model (GIM), which allows users to provide new forms of syntactic feedback on programs suggested by the synthesizer. This includes: (1) A GIM implementation for linear functional programs, which allows the user to approve or disapprove of specific sub-sequences of a candidate program, rather than just responding to it as a whole; and allows a synthesizer to present candidate programs with debug information. (2) A formalization of user-driven interactive synthesis, which allows us to crystallize the shortcomings of working with the holistic PBE model, and to show how our interaction model complements it. (3) A controlled user study showing that programmers have strong preference for using granular feedback instead of examples, and are able to provide granular feedback much faster.

• In Chapter 5 we formalize the iterative model presented in Chapter 4 and use it to prove properties on iterative synthesis sessions. This includes (1) A general model for iterative synthesis using the theory of abstract domains, (2) Convergence conditions for iterative synthesis sessions, based on properties of the predicates and user behavior, and (3) Insights about backtracking when a session can no longer converge.

• In Chapter 6 we collect recommendations for designers of future synthesis tools, as they stem from the results shown in previous chapters.

We believe this thesis highlights an important void in synthesis research, i.e., the interactivity of all synthesis tools, whether or not they are described as such. The conclusions drawn from the research in this thesis are important lessons for the field of program synthesis, and directions proposed in this thesis hold much potential for future research.

The work described in this thesis was published in [PIS18, Pel18, PRY18, PSY18].
Chapter 2

Preliminaries

This thesis addresses the synthesis of functional programs. In this chapter we provide the necessary background.

Functional programming  Functional programming is a programming paradigm that breaks a desired operation into a composition of functions, and as such also prefers immutable objects. A purely functional solution to a problem would be broken down into a stateless composition of functions \( h(g(f(x))) \). Functional programming coexists with object-oriented programming in languages such as Scala, in which case a popular paradigm is a functional composition of object methods, where \( f(self) \) is an object method of \( f \), which returns a second object (in order to support immutability) for which \( g \) is an object method, allowing the program \( g(f(self)) \), which returns a third object, and so forth.

Notation of functions  We interchangeably use the mathematical notation \( h(g(f(x))) \) for the functional composition called on object \( x \) and the Scala notation \( x.f.g.h \) (in Scala, a function application with no arguments does not require parentheses).

For a functional program \( m \), we denote \([m]\) as the function that the program computes. Formally, \([m] : D \to D \cup \{\perp\}\) maps every element \( i \) in the domain, \( D \), either to the element in \( D \) that the program outputs on \( i \), or to an error (compilation or runtime) \( \perp \notin D \).

Types  We assume a typed domain \( D \) over which programs operate, where each element in \( D \) has a type. Let \( T \) be the set of all types occurring in \( D \), and let \( \subseteq \) be the partial order of inheritance over \( T \). I.e., \( \tau_1 \subseteq \tau_2 \) if \( \tau_1 \) inherits \( \tau_2 \) or \( \tau_1 = \tau_2 \). We write \( \tau_1 \subset \tau_2 \) when \( \tau_1 \subseteq \tau_2 \) and \( \tau_1 \neq \tau_2 \).

The synthesis problem description  Readers familiar with software verification would most likely recognize the common verification problem \( \forall i. \varphi(i) \), where \( i \) ranges over possible program inputs and \( \varphi \) is a property to check (safety, liveness, termination,
etc.). In synthesis, the problem is commonly stated as $\exists m. \forall \iota. \varphi(m, \iota)$, where $m$ ranges over the domain of \textit{candidate programs}, and the synthesizer is tasked with finding one program that satisfies the desired property on all inputs. Different tools have varying ways to define the candidate program space. Since this space is huge even when considering a modest program size, sifting through it to find a single program with the property $\varphi$ is computationally hard.

**Vocabulary and the candidate program space** The candidate program space consists of programs of the form $\text{input}.f_1.\ldots.f_n$ (in Scala notation), or $f_n(f_{n-1}(\ldots f_1(\text{input})\ldots))$ (in mathematical notation), where each $f_i$ is a method from a predefined vocabulary $\mathcal{V}$. Object methods that accept arguments are handled by partially applying them with predefined arguments, such as constants, lambda functions or variables in the context, leaving only the self reference as an argument. Generally, the candidate program space includes every program in $\mathcal{V}^*$, but we notice that for some programs there are compilation errors as not all $f \in \mathcal{V}$ are applicable to all objects.

**Program semantics** In the most abstract sense, a program $m$ accepts input $\iota \in I$ and produces output $\omega \in O$. In programs that have effects on their environment (sending network packets, moving a robotic arm) the environment state can be folded into the input and output spaces; so for all purposes, we can assume a definition of program semantics as $[m] : I \rightarrow O \cup \{\bot\}$. The special value $\bot$ indicates abnormal behavior, which may be a run-time error (including abort) or non-termination. It means there is no execution of the program with the given input that reaches the designated “successful” exit point.

**Partial specification** Often, it is quite hard to describe the property $\varphi$ to the synthesizer precisely. Most synthesizers offer a domain-specific language for describing weaker properties in a way that both the user and the synthesizer can understand and (hopefully) the synthesizer can efficiently generate a corresponding program. We present two examples for such property domains.

- **Type-directed synthesis** is a sub-class of the synthesis problem where the specifications are in the forms of types: the types of the input variables and the expected output type. Likewise, the construction of programs is guided by derivation rules that are constructed from the typing rules. On a very basic level, the construction of well-typed programs is type-directed synthesis, but synthesizers often contain derivation rules that select a small number of operations to derive when adding operations or assigning parameters.

This class can be restricted to our setting of monotonic refinement if at each step the user can only (a) remove variables from the scope, (b) generalize an input
variable’s type to a super-type, or (c) concretize the expected return type to a sub-type.

- **Programming by Example (PBE)** Programming by Example is a sub-class of program synthesis where all communication with the synthesizer is via examples. The classic PBE problem is defined as a pair \((E, L)\) of initial examples \(E\) and target language \(L\), where each example in \(E\) is a pair \((i, o)\) of input \(i \in I\) and expected output \(o \in O\). Typically \(I\) and \(O\) are both \(D\), the domain of all values available to the language \(L\). The result of the PBE problem \((E, L)\) is a program \(m\), which is a valid program in \(L\) that satisfies every example in \(E\), i.e., \([m](i) = o\) for every \((i, o) \in E\). Since there might be more than one program \(m\) in the language \(L\) that matches all specifications, the iterative PBE problem was introduced. In the iterative model, each candidate program \(m_i\) is presented to the user, who may then accept \(m_i\) and terminate the run, or answer the synthesizer with additional examples \(E_i\) that direct it in continuing the search.
Chapter 3

Generating Tests by Example

In this chapter, we present JARVIS, a tool for generating property-based tests from unit tests. JARVIS performs very large generalizations from very few data points, which means the occasional manual intervention by the user was required. As such, JARVIS is a test case in simple, manual inclusion of the user in program synthesis.

3.1 Introduction

Parametric unit-tests [TS05a, TdH08, SBE08, TS05b, XTdHS09] are a well-known approach for increasing coverage and thus increasing confidence in the correctness of a test artifact. Parametric unit tests (PUTs) are also a common target of automatic test generation [FZ10, AGN+09] and unit test generalization [FZ11, TMX+11, CIvdP07]. A parametric unit test consists of a test body defining the parametric code to execute, and a set of assumptions that define the requirements from input values.

Parametric unit tests can either be symbolically executed or instantiated, which is the process of turning them back into unit tests [TS05a, TdH08]. One way to instantiate PUTs is to provide them with concrete values based on whitebox knowledge of the program under test [TS05b, YTP+16]. Another way is to provide a value generator for the parameters, usually hand-crafted by an expert, which generates appropriate values on demand. This type of test is called a property-based test (PBT) [FB97, CDG10, DWA+09, Hug07].

The paradigm of property-based testing [CH11, scaa, jsv, Pik14] defines the desired behavior of a program using assertions on large classes of inputs (“property”). To test the program, property-based testing generates inputs satisfying the precondition to check whether the assertion holds. Property-based testing is known to be very effective in checking the general behavior of the code under test, rather than just on a few inputs describing the behavior. It does this by describing the behavior as assertions over classes of input, generating random inputs from that class to check the assertion against. This has the advantages of increasing both instruction coverage and value coverage, and exposing bugs which may be hidden behind the selection of specific representative test
In this chapter we present a technique for automatic synthesis of PBTs—parametric unit tests, together with an appropriate value generator—from repetitive unit tests.

The value generators synthesized by our approach follow relationships captured by an abstract representation to explore values within the test’s input assumptions. In contrast to the assumptions of parametric unit tests which require a separate enumeration technique (e.g., based on whitebox guidance), abstraction-based generators contain nothing but the definition of the desired input space, and so can be sampled directly and repeatedly to provide a large number of additional values that satisfy the required assumptions.

Our approach generalizes existing unit tests by finding tests with a similar structure such that their concrete values can be over-approximated using an abstract domain. This allows us to use the executed code from the original test, as well as the oracle (assertion) of the test, and execute them with new concrete values. Our approach learns from both positive and negative test-cases (i.e., tests expected to succeed and fail, resp.), enabling a more precise generalization of tests. Specifically, it finds an over-approximation for the positive examples, while excluding any negative examples, and vice versa. In addition, our generalized tests preserve constraints inherent in concrete unit tests, such as types and equalities, which allow us to address the subtle nuances tested by them.

**Challenges**

To achieve our goal, we have to address the following challenges:

- Identify which tests, along with their oracles, should be generalized together to obtain parametric tests.

- Generalize matching tests to find an over-approximation that represents all positive examples but none of the negative ones. This will allow us to synthesize value generators that match the generalized tests.

**Existing Techniques**

Thummalapenta et al. [TMX+11] conducted an empirical study analyzing the cost and benefits of manually generalizing unit tests, and have shown that the human effort pays off in increased coverage and newly detected bugs. Shamshiri et al. [SJR+15] conducted a test of state-of-the-art test-generators and concluded that they do not create tests as meaningful as human-written tests, leading to the conclusion that basing generalization on existing tests will lead to better results. Fraser and Zeller [FZ11] create tests with pre- and postconditions on parameters, but do so by assuming a baseline version of the program and, in practice, incorporate its bugs into the tests. Francisco et al. [FLFC13] created PBTs for web services, but did so from a semantic description that had to be manually written for each web service. Loscher and Sagonas [LS17] improve upon PBTs with guided value generation, rather than simple random sampling.
Our Approach  The main idea is to leverage the repetitive nature of existing unit tests to automatically synthesize parametric tests and generators. Technically, we define a partial order on the set of tests, that captures the generality of the test data. This order allows our technique to use the same unit test as an example for several different PBTs, capturing different subtleties, and at the same time staves off over-unification of example sets that would yield meaningful results individually, but a non-informative generalization together. We use safe generalization [PSY16] to separate positive and negative examples.

Dividing Tests  Provided with individual example unit-tests to be used as a training set, we aim to divide them into sets to be abstracted, in order to create the smallest number of abstractions that are still meaningful, and can still be sampled. We then aim to determine how many value generators are to be created for each such abstracted region of the parameter space. The goal of the division is to create a set of value generators for the property-based tests that will be generated such that each abstracted region can over-approximate the maximal number of examples, and different value generators are created over the same region preserve the testing nuances seen in the original tests. The motivation is that a generator for a PBT must contain the constraints of the subtle cases that were selected by the programmer, to guarantee that these cases are covered in a non-negligible probability when the PBT is executed.

To support this goal, we define a partial order of generality between PBTs. This allows us to create a value generator for each testing nuance, and do so on the maximal number of examples that are compatible with this subtlety.

Safe Generalization of Tests  Given a set of compatible positive tests (expected to succeed) and negative tests (expected to fail), we wish to generalize them into a region that a PBT’s value generator can sample. To that end, we use an abstraction method for separating positive and negative examples, called Safe Generalization [PSY16].

Implementation  We present JARVIS (JUnit Abstracted for Robust Validation In Scalacheck), a tool that extracts repetitive tests from unit test suites, determines their place in the partial order, and synthesizes from them PBTs that generate inputs based on preserved properties. We test JARVIS on unit tests from Apache projects. We also show that sampling the abstracted over-approximations increases value coverage[Bal04, HSW99] of the exercised code while not losing instruction coverage. In addition, we demonstrate JARVIS’s ability to discover historical bugs when run on test suites in the previous versions.

Main Contributions  The contributions in this chapter are:

- An inclusion relation between parameterized tests that allows the sharing of
examples between different abstracted generators without hindering the ability to abstract.

- A technique that generalizes test values from individual unit tests into value generators for a PBT using safe generalization (separating positive and negative examples).

- A tool, JARVIS, that automatically synthesizes parametric tests, oracles and abstraction-based generators from unit tests, while preserving the subtle cases that are captured in these tests.

3.2 Overview

Unit tests are an integral part of the software development process. They are used to test small components of large software systems independently. Such components can typically receive many possible inputs, and in order to cover their different behaviors, a component is often run using the same test code with several different input values. In practice this leads to repetitive test code to exercise the same unit under test again and again. An initial study of the repetitiveness in the test suites of five large Apache projects (Commons-Math, Commons-Codec, Collections, Sling and Spark core) showed that of 13,359 total tests, 40% are not unique test scenarios, and 17% are repetitive by being written as an assertion called inside a loop. In some test files, all test code is non-unique either by virtue of repetition or loops. This means that repetition of individual tests is not only present but frequent.

However, these tests still use the same values every time the test suite is run. Running identical code with other possible values may reveal a bug, and new bugs may be introduced that will not be tested because of the test values are constant. In fact, tracing through the history of the testing code shows us many such cases: identical tests with a small change of constant values that were later added to represent a bug that has been discovered, and often has been in the code for a full version or more.

We set out to take repetitive test suites and synthesize from them testing properties for property-based testing. Once we have in our possession a parameterized test with an assertion to test its postcondition, as well as a set of values for the parameters labeled for expected success or failure of the test, we can use previous work [SA14, SG09, LM14, FL01, ECGN01] to learn a precondition on the data and convert it to a data generator for a PBT.

However, dividing test traces into compatible sets is not trivial. Tests may seem to be representing the same case but in their over-unification harm the abstraction. In addition there may be sets of tests that represent an interesting test case, such as equal parameters or a subtype being used, which should be preserved when sampling.

This chapter addresses the following problems:

1. Finding individual tests that can be generalized together (“compatible”);
2. Generalizing the tests into a property-based test that would cover a superset of the original tests; and

3. Creating abstraction-based value generators that will sample the abstraction while preserving testing nuances.

To solve 1 we define the notion of tests that are compatible—that test the same thing, and so have the same notion of correctness behind the examples. To solve 2, we use the notion of Safe Generalization in order to find an abstraction that will separate completely the example test cases that are expected to succeed from those that are expected to fail. Finally, to solve 3, we sample these abstractions in a constrained manner dictated by the original tests.

We demonstrate these steps on a real-world example taken from the Apache Commons-Math test suite.

The code segment in Fig. 3.1 depicts duplicate tests with different constant values in the class `PrecisionTest` in the Commons-Math project. We notice that the seemingly straightforward duplication is not exact duplication. For instance, the test in line 1 uses the same value twice, creating an equality constraint. In fact, in the larger file, there are several such tests, using different constants but repeating the value between the first and second parameter.

This means that there is an explicit intention to test the case where the two parameters are equal. Leaving this to chance while drawing reals would make getting two equal values highly unlikely, and the synthesized property would be skipping an intentional special test case if this is not performed. We therefore wish to generate as our output not one but two tests: one for the general case and one for the test with the equality constraint.

**Parameterized tests** Each test trace is turned into a parameterized test. In a parameterized test, constants are extracted and replaced by parameters of the same type. Parameter extraction takes into account constraints that exist in the concrete test, which means that if the same value appears more than once it will be extracted as the same parameter every time. For instance, `assertTrue(Precision.equals(153.0000, 153.0000, .0625));` (line 1) will be parameterized to `pt_1 = assert(Precision.equals(x, x, .0625));`
with types \( \text{type}(x) = \text{type}(y) = \text{double} \), and the parameter mapping of \( \{ x \mapsto 153.0, y \mapsto .0625, \text{res} \mapsto + \} \) is preserved, where \( \text{res} \) signifies the expected result of the assert. Similarly, lines 2-5 will all be parameterized into \( pt_2 = \text{assert}(\text{Precision.equals}(x, y, z)) \); with \( \text{type}(x) = \text{type}(y) = \text{type}(z) = \text{double} \), with four matching parameter mappings.

**Grouping parameterized tests into scenarios** Parameterized tests that test the same sequence of statements but for the different parameters are grouped together into *scenarios*. All parameterized tests in such a scenario would yield a property-based test that runs the same code, only with differently drawn values for the parameters. In Fig. 3.1, both the parameterized tests \( pt_1 \) and \( pt_2 \) are testing \( \text{assert}(\text{Precision.equals}(?, ?, ?)) \) and will be grouped into the same scenario.

All parameterized tests in the same scenario execute the same trace, or in other words test the same thing. A naive solution could use the parameterized data from all test traces belonging to a scenario, and simply perform the abstraction on them, generating a single property-based test for the entire scenario. However, because of the transition from constant values to randomly generated ones, information about the intent of the test is lost. E.g., if the parameterized test sends an integer to a double argument of a method, there is an intent for a number with no fractional part. If the parameterized test repeats a value throughout the test (e.g. between method arguments) there may be an intent for equality. In both cases, the chance of obtaining a value that fits the intention when drawing random values—e.g. from \( \mathbb{R}^3 \) in the case of Fig. 3.1—is slim at best.

A simple solution for this could be to keep the tests separated by the parameterized tests that contain them. This means all examples from tests that match \( \text{assert}(\text{Precision.equals}(x, x, y)) \) with \( \text{type}(x) = \text{type}(y) = \text{double} \) will be joined, separate from those that match \( \text{assert}(\text{Precision.equals}(x, y, z)) \) with \( \text{type}(x) = \text{type}(y) = \text{type}(z) = \text{double} \). This would generate an additional test forcing the equality of arguments, but would withhold from the unconstrained case with three parameters the additional data points that were separated out. Since both these parameterized tests call the same method, these data points contribute to the understanding of the method’s general behavior, and this would cause the generalization of the second test to learn from fewer samples.

**A hierarchy of tests** A more realistic solution is to abstract as many examples as can be safely unified together, and sample each abstracted region separately later. To do this, we create a hierarchy of parameterized tests based on their parameters. For each parameterized test, we may also consider the data from all the tests below it in the hierarchy. When creating abstractions for the scenario, we consider the maxima of the hierarchy, along with all additional tests that have propagated up to them. This shares as many examples as possible, while preventing over-unification.
To do this, we define an inclusion relation between parameterized tests belonging to the same scenario, based on the sequence of all parameter uses in the test trace. In our example, \( pt_1 \) has the parameter sequence \( x \cdot x \cdot y \) whereas \( pt_2 \) has the parameter sequence \( x \cdot y \cdot z \).

We will say \( pt_1 \) is a subtest of \( pt_2 \) because (i) every parameter in place \( i \) in the sequence for \( pt_1 \) has an implicit conversion to the parameter type of the parameter in place \( i \) in the sequence of \( pt_2 \), and (ii) any equality constraint in the usage sequence of \( pt_2 \) (i.e. the parameter is repeated between places \( i \) and \( j \)) is also present in the sequence of \( pt_1 \). In this case, (i) holds trivially as the types are the same, and (ii) holds because the constraints in \( pt_1 \) are relaxed to no constraints in \( pt_2 \).

Section 3.4.2 details the \( \sqsubseteq \) relation between two parameterized tests. Section 3.8 presents experimental data on the importance of using the hierarchical approach.

**Abstracting the test data**

Now that the parameterized tests have been ordered and their concrete samples shared, we can abstract the values of the maxima of the \( \sqsubseteq \) relation to a more general behavior. Earlier we parameterized the expected result of the trace with the assignment for \( res \), indicating whether the concrete test should succeed when tested with the constants in the current parameter assignment. This can be used as a label for the parameter assignments as positive or negative examples of the more general property, which we wish to abstract. The examples comprising \( pt_2 \) yield the following two sets:

\[
\text{Positive} = \{(153.0000, 153.0000, .0625), \\
(153.0000, 153.0625, .0625), \\
(152.9375, 153.0000, .0625)\}
\]

\[
\text{Negative} = \{(153.0000, 153.0625, .0624), \\
(152.9374, 153.0000, .0625)\}
\]

We are interested in finding an abstraction for the Positive and Negative sets which explains the partition above, and enables us to generate many more positive and negative examples. It is vital that the abstraction will create a clear-cut separation between the positive and negative examples, in order to ensure that the values drawn will be a superset of the existing examples. This is also a reason that having a large example set is important: having more examples helps grow the abstraction, and having more counterexamples will limit the positive abstraction from covering portions of the input space that should be negative.

To do this, we use the notion of Safe Generalization, and abstract both the positive and negative samples simultaneously, checking that the abstraction of positive examples has not grown to cover negative examples and vice versa.

If there are several maxima in the relation that are being abstracted separately,
we notice that the Negative set for each of them contains negative examples for the scenario behavior. This means each Positive set should be separated from all Negative sets, and vice versa. These additional points to be used as counterexamples will improve the separation.

In our case, the abstraction describing the positive examples is $|x - y| \leq z$, and its negation for the negative examples. Section 3.5 formally defines Safe Generalization and details the use of JARVIS’s template library.

In cases where the abstraction is performed on very few samples, there is a lot of room for error for any abstraction. In other programming by example tools such as [GHS12, LG14], the solution is to allow the user to mark the solution as incorrect and provide more examples. Section 3.7 discusses the reasons the abstraction may not be ideal and possible solutions.

**Sampling the abstraction** Once an abstraction is obtained for some set $PT = \{pt_1, pt_2, \ldots, pt_n\}$ of parameterized tests, we turn our attention to sampling the abstracted region, and to the preservation of testing nuances. Because we consider test cases written by the user a weighted sampling of the abstract behavior, we want to make sure we model the sampling of our PBTs in the same fashion.

To do this, we generate an abstraction-based value generator for each $pt_i \in PT$, which will practice constrained sampling, i.e., draw values from the abstract region under the parameter constraints of the parameterized test. Section 3.6 details the way value generators are created over the abstract region.

Finally, we synthesize a PBT to includes each value generator. Fig. 3.2 shows the resulting properties both the positive and negative data abstractions applied to the concrete samples in $PT = \{pt_1, pt_2\}$, sampled according to $pt_2$.

Running this property will test the parameterized test on hundreds of values each time. This means that values matching the expected behavior but not covered by the concrete tests will now be tested. This can find bugs that are simply not tested for, and if the test property is added to the test suite, can help stave off bugs that will be added in future changes to the code. Section 3.10 shows a case study of a historical bug in Apache Commons-Math that was found by using JARVIS on the library’s test suite in the version before the bug was corrected.

### 3.3 Property-Based Testing

In this section we introduce concepts used in this chapter, including property-based testing and value-based coverage metrics.

**Unit test** A unit test consists of stand-alone code executed against a Unit Under Test (UUT), the result of which is tested against an oracle (an assertion) for correctness. In practice, the code exercising the UUT often targets a small unit, and the oracle is
```scala
val gen_double_1_pos = for{
  y <- Arbitrary.arbitrary[Double].map(Math.abs);
  x <- Arbitrary.arbitrary[Double];
  z <- Gen.choose[Double](x - y, x + y)
} yield (x, y, z)
forAll (gen_double_1_pos) { _ match {
  case (d1: Double, d3: Double, d2: Double) =>
    Precision.equals(d1, d2, d3)
}
}
val gen_double_1_neg = for{
  y <- Arbitrary.arbitrary[Double].map(Math.abs);
  x <- Arbitrary.arbitrary[Double];
  z <- Gen.oneOf(
    Gen.choose[Double](Double.MinValue, x - y)
      .suchThat(_ < x - y),
    Gen.choose[Double](x + y, Double.MaxValue)
      .suchThat(_ > x + y)
  )
} yield (x, y, z)
forAll (gen_double_1_neg) { _ match {
  case (d1: Double, d3: Double, d2: Double) =>
    !(Precision.equals(d1, d2, d3))
}
}
```

Figure 3.2: The ScalaCheck properties synthesized from the test traces shown in Fig. 3.1.

implemented by a set of assertions testing the state and output of the unit test code. Unit testing tools such as JUnit [jun] and NUnit [nun] provide an environment that can execute an entire test suite of unit tests.

**Property-based test** PBTs consist of test code and an oracle that are defined over parameterized classes of values. For that class of values, the PBT is phrased as a “forall” statement or axiom on the behavior of a component. This means PBTs mirror not a specific code path, but the specifications of the UUT. For example, a simple property on strings would specify that $\forall s_1, s_2, \text{len}(s_1 \cdot s_2) = \text{len}(s_1) + \text{len}(s_2)$.

A property-based test is comprised of two parts: the test body and oracle, which are the code operating on the UUT and the boolean statement which must hold, in this example concatenating and testing the length of strings; and the generator, which defines the class of inputs on which the PBT is defined, in this example any two non-null strings.

This is similar to the way parameterized unit tests [TS05a] are defined. However, PUTs define the input class by assumptions on the parameters. This means that in order to run as tests in the test suite, PUTs need to be run through a solver or a symbolic execution of the UUT in order to be instantiated with values for the parameters, methods
which are usually whitebox. The instantiated parameters are added to the test, which is then transformed into a conventional unit test. Barring a re-run of the solver, the values on which the resulting tests are run are constant.

In contrast, PBTs are intended for execution of the test body on a random sample of values that are drawn from the generator. The generator, rather than describing the input class as a boolean formula (i.e., the conjunction of all assumptions) that filters inputs, defines concretely a portion of the input space from which values can be drawn.

A test using the generator can be added as-is to a test suite using PBT frameworks such as QuickCheck [Hug07, CH11], PropEr [PS11], JSVerify [jsv] and ScalaCheck [sca] that include an initial implementation for the building blocks of generators, such as ScalaCheck's Gen.choose used in Fig. 3.2.

It has been shown [TMX+11] that test parametrization is worthwhile in terms of the human effort it requires and the bugs that are detected. It can be extrapolated that PBTs, for which it is easier to draw a large set of test values, would be a worthwhile substitute.

3.4 Compatible Tests

3.4.1 From Test Trace to Parameterized Test

In this section, we formally describe how different unit test within a single test suite can be viewed as a repetition of the same test with different parameters. We then continue and formalize what we consider as subtle cases, or testing nuances, appearing in such a group of repetitive tests, and explain how our technique still preserves them.

The first step of our technique is to identify test traces in the original test suite. A test trace is a sequence of (not necessarily adjacent) statements ending with a single tested assertion, that can be executed sequentially.

For example, lines 3 – 4 of Fig. 3.3 form the test trace `Interval interval = new Interval(2.3,5.7); assertEquals(3.4,interval.getSize());`. Each line in Fig. 3.1 forms its own test trace, e.g. `assertTrue(Precision.equals(153.0,153.0,.0625));` is formed by line 1.

To handle the many test traces in a library’s test suite, we must group them into sets of tests that are compatible for a common abstraction. To this end, we first normalize them and create tests that do not use any specific constant values. This normal form is called a parameterized test. Technically, a parameterized test obtained from a test trace contains the same statements as in the test trace, where constant values are replaced by an uninterpreted parameter of the same type as the constant. Moreover, if the same constant appears multiple times in the test trace (at different locations), all occurrences are replaced by the same parameter. Finally, specific assertions such as assertTrue or assertFalse are replaced with a general assert command.

As seen in Section 4.2, the test trace in line 1 of Fig. 3.1 is parameterized as
\( pt_1 = \text{assert}(\text{Precision.equals}(x, x, y)); \) with types \( \text{type}(x) = \text{type}(y) = \text{double} \). Similarly, the test trace in lines 3–4 of Fig. 3.3 is parameterized as \( \text{Interval interval} = \text{new Interval}(x, y); \text{assert}(z==\text{interval.getsize()}); \) with \( \text{type}(x) = \text{type}(y) = \text{type}(z) = \text{double} \).

Note that while a concrete test trace holds correctness information (i.e., the desired result of the assertion on a concrete execution of the trace), a parameterized test no longer encodes any such information. The expected result of the assertion is stripped along with the constant values, as it depends on them: the exact same parameterized test might be a positive test on one set of values and a negative test on another.

The relation between a parameterized test and a test trace from which it was originated, relies on the following definition:

**Definition 3.4.1 (Parameter mapping).** A parameter mapping for a parameterized test is a function \( f \) that maps every parameter \( x \) to a constant \( c = f(x) \) s.t. \( \text{type}(x) = \text{type}(c) \). Additionally, \( f \) maps a new variable \( \text{res} \) to \{+, −\}.

Essentially, a parameter mapping is a function that reproduces the original test trace from a parameterized test. The role of \( \text{res} \) in the definition above is to represent the type of the assertion (positive or negative). We can think of a test suite as a set of parameterized tests, where each such parameterized test is equipped with a set of parameter mappings \( F = \{f_1, \ldots, f_n\} \). Applying each \( f_i \) to \( pt \) will yield a concrete test trace \( t_i \).

### 3.4.2 Separation

Section 3.5 will explore the abstraction mechanism, but it is easy to see that an abstract representation could be more accurate when working on as large a number of examples as possible. An abstraction that only takes into account the values obtained by the parameter mappings attached to a certain parameterized test may result with a small number of concrete samples. This may yield an abstract representation which is too coarse. Even worse, the abstract representation may provide no generalization.

To address this, we introduce another definition relying only on the statements in the test trace:

**Definition 3.4.2 (Scenario).** A scenario \( S \) is a set of parameterized tests which execute the same sequence of statements, differing only by their parameters. The code of a scenario \( S \), is the sequence of statements mutual to all parameterized tests in \( S \), after discarding parameter information. We say that a parameterized test \( pt \) belongs to a scenario \( S \) if the code of \( S \) is obtained by discarding \( pt \)’s parameter information.

Continuing our example with \( pt_1 = \text{assert}(\text{Precision.equals}(x, x, y)); \) if \( S \) is the scenario to which \( pt_1 \) belongs, then the code of \( S \) is the statement \( \text{assert}(\text{Precision.equals}(?, ?, ?)); \) (without parameter information).
The unification of parameterized tests into scenarios is driven by the fact that despite
the different parameter mappings, they are all running the same code (It is important
to note that method overloading information is not discarded.)

Next, we formalize subsumption between parameterized tests of the same scenario.
These definitions will allow us to increase the number of parameter mappings that can
be attached to a single parameterized test.

To define subsumption, we wish to compare two parameterized tests from the same
scenario and assess their generality. To do that, we need to compare the parameter uses
in the parameterized test in sequence. We therefore rely on the following definition:

Definition 3.4.3 (Sequence of parameters). Given a parameterized test $pt$, let $\text{params}(pt)$
be the sequence of parameters across all statements in the parameterized test $pt$ (with
repetitions).

This notion is needed so that we may compare two parameterized tests in the same
scenario with a different number of parameters or with equality constraints in differ-
ent places in the test trace. E.g., for $pt = \text{foo}(x, y); \text{assert}(\text{bar}(x, z))$; we have
$\text{params}(pt) = x \cdot y \cdot x \cdot z$.

Definition 3.4.4 (generality of parameterized tests, $\sqsubseteq$). For two parameterized tests
$pt_1, pt_2$ with $\text{params}(pt_k) = x_1^k \cdot \ldots \cdot x_n^k$ for $k \in \{1, 2\}$, both belonging to the same scenario
$S$, we say that $pt_1 \sqsubseteq pt_2$ if $\forall i, j \in \{1 \ldots n\}$:

1. $\text{type}(x_i^1) \sqsubseteq \text{type}(x_i^2)$ (we use the standard notion of this relation, e.g. $\text{int} \sqsubseteq \text{double}$,
   $\text{String} \sqsubseteq \text{Object}$.)
2. $\text{name}(x_i^2) = \text{name}(x_j^2) \Rightarrow \text{name}(x_i^1) = \text{name}(x_j^1)$

The definition above allows us to create a parameter mapping $f_2$ for a parameterized
test $pt_2$ from a parameter mapping $f_1$ for parameterized test $pt_1$, such that $pt_1 \sqsubseteq pt_2$.
We do this by defining the result of $f_2$ for every $x_i^2 \in \text{params}(pt_2)$ by $f_2(x_i^2) = f_1(x_i^1)$.

The implication of the correctness of behavior described by all parameterized tests
in a scenario is that all parameter mappings in a scenario can and should be abstracted
together. However, creating a single abstraction for the entire scenario will create a
unification problem.

Example 3.4.5. Let us consider three parameterized tests that have several parameter
mappings each: $pt_1 = \text{int prev = x.size(); x.add(y); assert(x.size() == prev + 1)}$; with $\text{type}(x) = \text{List<String>}$ and $\text{type}(y) = \text{String}$, $pt_2$ is identical to $pt_1$
except for having $\text{type}(x) = \text{ArrayList<String>}$, and $pt_3$ is identical to $pt_1$ except for
having $\text{type}(x) = \text{Set<String>}$. Since add and size are methods on $\text{List}$ and $\text{Set}$’s
shared parent interface $\text{Collection}$, $pt_1, pt_2$, and $pt_3$ all belong to the same scenario.

Since $\text{params}(pt_1) = \text{params}(pt_2) = \text{params}(pt_3) = x \cdot x \cdot y \cdot x$, and since $\text{ArrayList}$
is a subtype of $\text{List}$, but $\text{Set}$ and $\text{List}$ only share a common ancestor, we see that
$pt_2 \sqsubseteq pt_1$, and $pt_3$ is incomparable with both.

26
We notice that even though $pt_1$ and $pt_3$ are incomparable, there exists a parameterize

test $pt_4$, with the same test code and $type(x) = \text{Collection<String>}$ and $type(y) = String$ for which $pt_1 \sqsubseteq pt_4$ and $pt_3 \sqsubseteq pt_4$.

If we aim to abstract $pt_4$, we can see that our unification problem is twofold. First,

we now need to abstract (and later generate values for) collections in general, not just

lists and sets, from concrete data that only includes lists and sets. We also see that

there is a difference in behavior between sets and lists in this test code which needs to

be captured by the abstraction: for a set, $res \mapsto true$ only if $y$ is not already a member

of the set, whereas for a list (ArrayList or otherwise), $res \mapsto true$ always. This problem

is made even worse in cases where the shared ancestor is $Object$.

In order to avoid these problems we set a unification rule as follows:

**Definition 3.4.6 (Abstraction candidates).** Let $T \subseteq S$ be the set of parameterized
tests in a scenario $S$ such that for every $pt \in T$, $\neg \exists pt' \in S. pt \sqsubseteq pt' \land pt \neq pt'$.

We define the abstraction candidates for $S$ to be the sets of parameter mappings

$AC_S = \{\{f \in pt' \mid pt' \sqsubseteq pt\} \mid pt \in T\}$. When performing abstraction, each $s \in AC_S$

will be abstracted on its own.

In other words, given the DAG defined by the $\sqsubseteq$ relation, we create an abstraction for
every root $pt$, including with it the parameter mappings of every parameterized test
reachable from $pt$. This means we only create abstractions for parameterized tests that
exist “in the wild”, whilst reusing as many test traces as possible in order to abstract
them.

### 3.5 Abstracting the test data

In the following section, we abstract each of the sets of examples in $AC_S$.

Once a parameterized test has its final set of concrete test traces, the $res$ parameter

can be used to divide them into positive and negative samples. For instance, the

parameterized test for $\text{Precision.equals}(x, y, z)$ with $type(x) = type(y) = type(z) = double$

from Fig. 3.1 has the following data:

$$C^+ = \{(153.0, 153.0, .0625), (153.0, 153.0625, .0625), (152.9375, 153.0, .0625)\},$$

$$C^- = \{(153.0, 153.0625, .0624), (152.9374, 153.0, .0625)\}.$$

**Safe Generalization** We are interested in an abstraction in some language that

would be a Safe Generalization [PSY16], or an abstraction that provides separation from

a set of counterexamples. Safe Generalization is defined as an operation that further
generalizes a set of abstract elements $A$ from an abstract domain [CC77] into another

set of abstract elements, $A'$, while avoiding a set of concrete counterexamples $C_{cex}$, and

provides the following properties:
1. **Abstraction**: $A'$ contains every concrete element that is abstracted by $A$ (even though $A \preceq A'$ may not hold)

2. **Separation**: No $c \in C_{cex}$ is abstracted by $A'$

3. **Precision**: Generalization is a direct result of the elements in $A$.

as well as a strive for **maximality** that is not relevant for this use. We wish to generate two properties for the parameterized test that we are abstracting: one expecting the test to succeed, and one expecting it to fail. The code in these two properties is the same except for a negation of the assertion, but they require different data generators. In order to create these two generators, we need two abstractions, $A^+$ for the positive examples of the parameterized test, and $A^-$ for the negative.

It is important to notice that, when a scenario has multiple abstraction candidate sets, they are still all representing the same behavior in the code under test, which means they are influenced by the counterexamples in the other sets as well. Specifically, while the positive examples were separated by the unification rule, and should not be abstracted together, they should still be separated from every negative point in the scenario, as they all represent some negative case for the same code. This applies symmetrically to the negative points.

We therefore define for an abstraction candidate $a \in AC_S$ the following example sets:

$$
C^+ = \{ f \in a | f(res) = + \} \quad C^- = \{ f \in a | f(res) = - \}
$$

$$
C^+_{cex} = \bigcup_{b \in AC_s} \{ f \in b | f(res) = - \} \quad C^-_{cex} = \bigcup_{b \in AC_s} \{ f \in b | f(res) = + \}
$$

and attempt to attain the separating abstraction for $A^+$ from $(C^+, C^+_{cex})$ and for $A^-$ from $(C^-, C^-_{cex})$.

It is clear that not every abstraction language will be able to accommodate this requirement. In addition, when there are few samples, many different elements from each language may fit, and we are not necessarily interested in the most precise one, which means we will need to relax the precision requirement of Safe Generalization.

In some domains such as Intervals, we are able to easily compute Safe Generalization using algorithms such as Hydra [Mur87], but we may still wish to perform a controlled loss of precision on the result. In other domains, computing Safe Generalization will be doubly exponential. Instead, we utilize a Safe Generalization relation, denoted $SG(C, C_{cex})$, which includes the set of abstractions that are safe generalizations for $(\{ \beta(c) | c \in C \}, C_{cex})$, where $\beta$ is the abstraction function for a single concrete element. $SG$ relaxes the precision requirement, allowing abstractions to be included in $SG(C, C_{cex})$ even for very small example sets $C, C_{cex}$.

In theory we would construct $SG$ over every available abstraction. In practice, we use $SG$ to test a set of given abstractions.
**Abstraction templates**  In order to select an abstraction language and an element of that language, JARVIS contains a library of abstraction templates, such as $|x - y| \leq z$, $x \in [a, b]$, etc. As previously shown by the FlashFill project [SG15], the case of learning from few examples (in FlashFill, often only one example) requires the notion of ranking the possible programs, or in our case, possible abstractions, so that correct programs will be ranked higher than incorrect ones, and likely programs higher than unlikely ones. While in [SG15] this ranking is learned from examples, in our implementation the templates have a predefined ranking that is applied for all instantiated abstractions that hold for all samples.

Every template $t$ of the template library is instantiated, and in the case of templates such as $x \in [a, b]$ or $|a \cdot x - y| \leq b$, the parameters are selected based on the existing samples. Templates are instantiated in pairs, one as an abstraction for the positive examples and one for the negative. The result is $A = \{(A^+, A^-) | (A^+, A^-) \in SG(C^+, C^-_{cex}), (A^-, A^+) \in SG(C^-, C^+_{cex})\}$. We then select from $A$ the highest ranking $(A^+, A^-)$, and create code sampling them as the generators for the properties.

This means the template library can be extended to include more abstractions, and the ranking can be modified to better suit a specific project or domain.

### 3.5.1 Handling Impreciseness

The abstractions we use are conservative, and overapproximate the concrete data that they abstract. On the one hand, this guarantees that cases that are present in the original unit test will be included in the generated PBTs. However, in some cases, even the best abstraction available in the template library will be too conservative, and also represent data points that will fail the PBT. This can happen for one of two reasons:

- The abstraction itself is not precise enough (e.g. a single interval, when the data requires a disjunctive abstraction, or a set of intervals).
- The number of examples is too low to precisely generalize from (e.g. generalizing from two examples, there is not enough data to reduce the set of abstraction templates).

Both cases require manual intervention: in the first case, the user can provide a finer abstraction, in the second case she can provide more examples, and in either case she can manually edit the resulting tests.

### 3.6 Sampling from the Abstraction

We now wish to sample the abstraction that was created in the previous section. When creating the abstraction-based value generators that will sample the abstraction, we take our cue from the original test traces and their parameterized tests. We consider the original tests written by the programmers to be a weighted sample from the region of the domain that is described by the “true” precondition of the tested behavior. That
is, the user has already selected points that they deem important. We therefore wish to preserve them.

We have created an abstraction of each region – an underapproximation of the positive and negative regions for each of the maxima of the $\sqsubset$ relation. We now wish to generate property-based tests, or in essence, to generate code that will sample from the concretization of our abstraction. The sampling component of the code in Fig. 3.2 is shown in lines 1 – 6 and 11 – 17. It is composed of the representation of the space and types of the variables to be sampled into.

In this section we describe the creation of such sampling for the abstractions we performed.

**Sampling based on user-encoded testing nuances** We notice that we may wish to sample each abstracted region more than once. Since the constraints of parameterized tests lower in the hierarchy represent constrained values sampled by the user, we wish to cover them in our generated sampling. Let us examine the parameterized test $pt_1 = \text{assert(Precision.equals(x, x, y));}$ with $\text{type}(x) = \text{type}(y) = \text{double}$ from Section 4.2. It is sampled out of the region abstracted for the entire scenario $S$ containing $pt_1$ as well as other tests for $\text{Precision.equals}$. Abstracting the topmost parameterized test by the $\sqsubset$ relation yields an abstraction in $\mathbb{R}^3$. When sampling $(x, y, z) \in A \subseteq \mathbb{R}^3$ the odds of satisfying the constraint in $pt_1$, i.e., $x = y$, are infinitesimal. If we wish to preserve the constraint entered by the user, we must sample the special case in which $x = y$ on its own.

**Sampling the constraints** In order to sample each set of constraints on its own, we create a sampling component as follows: for every $pt \in S$, we create a sampling component over each abstraction for an abstraction candidate $s$ for which $pt$ has contributed its parameter mappings. If, according to the $\sqsubset$ relation, there are positive parameter mappings that apply to $pt$, a sampling component for the positive abstraction will be generated. Likewise for negative parameter mappings.

For each region sampled, the constraints of $pt$ are added to the restrictions on the domain. For example, when sampling the region $|x - y| \leq z$ for $pt_1$ seen in Section 4.2, the new sampling constraints are $|x - y| \leq z \land x = y$, or $0 \leq z$, sampling $(x, z)$ out of this region s.t. $\text{type}(x) = \text{type}(z) = \text{double}$.

**Sampling guarantee** Finally, we formulate our guarantee for points that will be sampled:

**Claim 3.6.1.** Let $T$ be a set of test traces from the same scenario $S$, $|T| \geq 2$. For each $t \in T$, if $\exists t' \neq t$ s.t. $PT(t) \sqsubset PT(t')$, then

1. $t$ will be used in an abstraction, and

2. $PT(t)$ will be used to create an abstraction-based value generator for a PBT.

30
This allows for a maximal reuse of examples for abstraction, and on the other hand, the sampling of all special cases that are abstracted.

### 3.7 The role of the user

Programming-by-example (“PBE”) solutions that work from a small number of examples have had to deal with the problem of selecting the wrong program from their search space. The common solution to this in these tools is to ask the user for more examples until the program that has been found is correct, or more specifically, produces correct results for all the available data. If we consider abstraction as a straightforward programming-by-example program, where the learned program is a function from the input space to a boolean In our case, it is easy to see why the abstraction problem here is comparable. However, there are two challenges that the abstraction problem has and normal programming by example does not: first, it is not enough to find a program that is correct for only the data that is provided by the user, but rather a program (abstraction) that is correct for the entire space is needed. This means it that any program but the correct program needs refinement. Second, the addition of another example to the set the abstraction is learning from is not trivial.

Offering a new example is sometimes time-consuming on its own. While string processing PBE tools only ask the user to perform the desired transformation on another input, our case asks the user to be familiar with both the code under test’s specifications and the technical limitations. For example, the case of finding the right $\mathbb{R}^n \rightarrow \{true, false\}$ for a mathematical function becomes tricky because of floating point arithmetic. The example in Section 3.10.2 shows a case where the ideal abstraction based on the specification, or on the understanding of the probabilistic functions being tested has to be reduced because each of the parameters must be limited to a range where floating point arithmetic does not cause such a loss of precision or even a complete zeroing out of all computations.

In cases such as these—and other, simpler ones, as well—each new unit test provided as an example must be programmed, tested to make sure it is correct and does not trip any undocumented technical limitations, and debugged if it does not produce the desired results. This means that if, as can often happen in PBE tools, the user is required to add two or three more examples, each one requires a repetition of the manual programming and debugging effort.

But, since JARVIS differs from other PBE tools in the type of effort required, namely programming effort rather than just applying the spec to more examples, it is not unreasonable to turn this effort to simply editing the PBT’s data generator if necessary. The code of the data generator is simple code, and executing the PBT provides examples that can be debugged to understand any unexpected failures, which may result from either the data generator still not representing the correct program, or from a hidden bug surfacing.
In addition, if we take into account the fact that this development and debugging effort leading to a correct data generator results in a PBT that can be added to the test suite, we can see that the return on the manual effort, an effort that is not as great as that described in [TMX+11] will reap at least the same returns, which they have shown to be highly profitable.

3.7.1 The role of the user in creating regression tests

If the code of the library under test is available, rather than just its interface, it is possible to perform an active-learning abstraction [PSY16], which generates more points that may improve the abstraction, tests them against the library and labels them as positive or negative. This means an example set much larger than that originally appearing in the unit test can be generated, making the abstraction far more accurate. Tools like [PSY16] direct point selection so that each point sampled increases confidence in the abstraction. Moreover, it would allow JARVIS to create property-based tests even for the test scenarios that have a single example (as can be seen in Table 3.1, this can be as many as 90% of the test scenarios).

However, this would expose JARVIS to the same problem inherent in testing tools that treat an existing version of the code as a regression baseline, such as [GSWH15, FZ11]: accepting as fact existing undesirable behavior that exists in the baseline version. However, since the output of the tool in this case would be a code description of the correct and incorrect input cases, it can be manually inspected by a developer for any undesirable behaviors manifesting in the property.

3.8 Experimental evaluation

We implemented JARVIS to operate on JUnit test suites written in Java and to synthesize ScalaCheck PBTs. Scala has a seamless interoperability with Java [OSV11], which means properties for ScalaCheck, which are written in Scala, can mimic completely the functionality of the original test traces. JARVIS uses the Polyglot compiler [NCM03] and the ScalaGen [scab] project to translate test traces from Java to Scala, and they are then paired with generators from the selected abstraction which are outputted directly to Scala. Template instantiation is aided by the Z3 SMT solver [DMB08].

We ran JARVIS on the test suites of several open source libraries. We tested whether the hierarchy and unification rule of abstraction candidates are relevant to real-world test suites.

3.8.1 Examining Apache test suites

Table 3.1 shows the result of running JARVIS on the test suites of 12 Apache Commons projects. This summary of JARVIS’s ability to unify shows us several things in regard to the problems it addresses:
### Table 3.1: Scenario makeup of the JUnit test suites of Apache-Commons projects. Repeating scenarios are those with the number of concrete test traces greater than 1.

<table>
<thead>
<tr>
<th>Library</th>
<th>avg size</th>
<th>repeating scenarios</th>
<th>no. of traces per scenario</th>
<th>no. of PTs have multiple PTs</th>
<th>Height roots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons-CLI</td>
<td>4.2</td>
<td>38.3%</td>
<td>3.5</td>
<td>14</td>
<td>1.067</td>
</tr>
<tr>
<td>Commons-Codec</td>
<td>2.2</td>
<td>38.2%</td>
<td>3.4</td>
<td>8</td>
<td>1.088</td>
</tr>
<tr>
<td>Commons-Collection</td>
<td>2.1</td>
<td>13.3%</td>
<td>2.8</td>
<td>5</td>
<td>1.133</td>
</tr>
<tr>
<td>Commons-Configuration</td>
<td>3.5</td>
<td>14.6%</td>
<td>3.7</td>
<td>15</td>
<td>1.015</td>
</tr>
<tr>
<td>Commons-CSV</td>
<td>3.2</td>
<td>27.8%</td>
<td>2.0</td>
<td>2</td>
<td>1.1</td>
</tr>
<tr>
<td>Commons-Email</td>
<td>2.3</td>
<td>60.0%</td>
<td>2.7</td>
<td>5</td>
<td>1.1</td>
</tr>
<tr>
<td>Commons-IO</td>
<td>2.9</td>
<td>23.3%</td>
<td>3.6</td>
<td>14</td>
<td>1.069</td>
</tr>
<tr>
<td>Commons-JEXL</td>
<td>4.6</td>
<td>25.0%</td>
<td>2.4</td>
<td>4</td>
<td>1.037</td>
</tr>
<tr>
<td>Commons-Math</td>
<td>4.3</td>
<td>19.7%</td>
<td>5.5</td>
<td>37</td>
<td>1.273</td>
</tr>
<tr>
<td>Commons-Pool</td>
<td>3.9</td>
<td>33.3%</td>
<td>4.1</td>
<td>45</td>
<td>1.182</td>
</tr>
<tr>
<td>Commons-Text</td>
<td>2.5</td>
<td>36.7%</td>
<td>2.0</td>
<td>15</td>
<td>1.032</td>
</tr>
</tbody>
</table>

Identifying tests In all projects, the average length of extracted test traces is over two statements. This shows JARVIS identifies and extracts elaborate tests.

Repetition of tests The data shows that there is, in fact, enough repetition of tests to justify test generalization. In the project with the least amount of repetition, Commons-Collection, only 13% of the scenarios contain more than one concrete test trace, but in half the projects this number is over 30%.

Existence of constraints When examining the scenarios that contain more than one test trace, we see that this is, on average, 9% of scenarios, with as many as 9 separate parameterized tests, or sets of constraints, in the same scenario. These are all user-encoded sampling constraints that would see their probability plummet without a specific sampler generated for them.

Importance of the unification rule In scenarios that have multiple parameterized tests, we see that the number of roots in the hierarchy DAG (i.e., incomparable maxima of the relation) is, on average, 1.35. This means that it is not infrequent to have scenarios where the least upper bound of two or more parameterized tests does not occur in the test traces. Since this is a frequent occurrence in the real world, we deem it frequent enough to address the issues that arise from over-unification within the scenario (as described in Section 3.4.2).

### 3.9 Instruction and value Coverage

Coverage metrics Test generation papers often use instruction or branch coverage as their guiding metric. However, when approaching a codebase with an existing test suite, these are already high. Additionally, as has been shown [Bal04], even 100% instruction coverage is no guarantee for a bug-free program. The values selected have a meaningful effect on actually finding faults in the program.
This has led to the development of several value-related coverage metrics. *Parameter value coverage* and *parameter value interaction coverage* \[SBV^{+08}\] counts the parameter values or parameter combinations that have not previously appeared. *Predicate coverage* \[Bal04\] aims to divide the program into equivalence classes of observable states and capture more complex data relationships needed to cover all reachable observable states. Likewise, *logical coverage* \[AOH03\] aims to find values that will cover predicates or clauses of the logical expressions in the program. *Boundary value coverage* \[HSW99\] aims to test the values at the boundary values of predicates on which the program branches, thereby aiming at off-by-one errors and similar bugs.

Most of these metrics are still anchored in the structure of the program. The implications of this are threefold: (i) blackbox methods of test generation cannot be driven by these metrics, as they would need access to the predicates and branches, (ii) they are still not sufficient in themselves to create a bug-free program, and (iii) tests generated by them will not suffice as regression tests for code changes. Consider, for instance, a program statement that might divide by zero. If there is no branch on the value of the divisor, tests generated to maximize the above metrics will have no reason to try two values – and specifically, zero. Additionally, consider a generated test suite, which is being used for regression testing while the code is modified. It may have originally had high coverage, but it is easy to insert new branches that were not considered in generation and so will now not be visited.

Because of these issues, we opt for a combination of two simple metrics: parameter value coverage, combined with instruction coverage. This allows us to perform blackbox synthesis of the tests – meaning the library’s code need not even compile, as long as the tests do – and aim for the spirit of PBTs, making axiomatic claims about the UUT that can be used for regression testing once the code has changed.

We tested two additional coverage questions:

1. Is instruction coverage preserved when testing with PBTs generated by JARVIS?

2. Is parameter value coverage increased when testing with PBTs generated by JARVIS?

Table 3.2 shows the coverage figures for repetitive unit tests from the Apache Commons-Math and Commons-Lang projects. For each test, it indicates its method of origin, or class of origin if there are several methods, the space of parameters the method accepts, and instruction coverage and parameter value coverage figures for both original JUnit tests and Scala PBTs synthesized by JARVIS. Instruction coverage figures were collected by JaCoCo \[jac\]. Parameter value coverage figures were collected from the PBTs by running it with the ScalaCheck default of 100 tests each, conserving the values sampled by the generator.

We notice that while instruction coverage was generally not improved by the synthesized PBTs, it was at least as great as the instruction coverage of the conventional
Table 3.2: Instruction coverage (IC) and parameter value coverage (PVC) figures for tests from Apache test suites and the PBTs generated for them by JARVIS.

JUnit tests in every case but one. In addition, the parameter value coverage for the PBTs was, on average, 26 times greater than the parameter value coverage for the JUnit tests. Parameter value coverage for the PBTs dropped below that of the JUnit tests in two tests, which we describe below.

**IntervalTest**  The parameter value coverage for the IntervalTest tests, which are shown in Fig. 3.3 is only 2, the same as the coverage of the JUnit test. This is because the PBT generated for this test (shown in Fig. 3.4) has found a bug. Because this bug is frequently occurring in terms of values in the parameter space of the function, the test failed, on average, on the second value generated. This is also the reason this test has managed to increase instruction coverage by two orders of magnitude: it has exercised code not previously seen by the conventional unit test. This bug is discussed in greater depth in Section 3.10.1.

**isPrintable and toIntExact**  In two tests, isPrintable and toIntExact, the conventional unit test’s parameter value coverage was greater than that of the synthesized PBTs. In both of these cases, the test function is run inside a loop on a large number of values, while each PBT’s execution was limited to 100 iterations.

While examining the PBT resulting from isPrintable, we noticed what appeared to be an inconsistency with the conventional unit test values in the JARVIS abstraction. Upon closer inspection, a copy-paste bug was revealed in the test code, testing some values with the wrong method. The test was fixed, and the commit was accepted by the Commons-Lang project\(^1\).

\(^1\)http://github.com/apache/commons-lang/pull/230
@Test
public void testInterval() {
    Interval interval = new Interval(2.3, 5.7);
    Assert.assertEquals(3.4, interval.getSize(), 1.0e-10);
    Assert.assertEquals(4.0, interval.getBarycenter(), 1.0e-10);
    Assert.assertEquals(Region.Location.BOUNDARY, interval.checkPoint(2.3, 1.0e-10));
    // Other asserts on properties of interval
}
// ...
@Test
public void testSinglePoint() {
    Interval interval = new Interval(1.0, 1.0);
    Assert.assertEquals(0.0, interval.getSize(), Precision.SAFE_MIN);
    Assert.assertEquals(1.0, interval.getBarycenter(), Precision.EPSILON);
}

Figure 3.3: Unit test code for the Interval class from the Apache Commons-Math project

testMinMaxDouble  The test testMinMaxDouble (which is actually two tests, for
FastMath.min and FastMath.max) is the only one in which the instruction coverage
of the PBT is lower than that of the JUnit test. This is because the conventional unit
test tests, among other values, Double.NaN, while the abstractions in JARVIS do not
abstract or create generators that can sample Double.NaN. As NaN (not a number) is
often handled in a separate code path, this leaves uncovered code when compared to
the unit test.

Conclusion  Having examined these special cases, we can answer questions 1 and 2
in the affirmative: instruction coverage is preserved and parameter value coverage is
increased in most cases by JARVIS.

3.10 Discovering Bugs: A Case Study

In this section we review two historical bugs in Apache Commons-Math that we
discovered by running JARVIS on the unit test suite for the version before the bug was
fixed.
val gen_double_1_pos = forall {
  x <- Gen.posNum[Double];
  y <- Arbitrary.arbitrary[Double];
  z <- Gen.oneOf(y-x, y+x).
suchThat (t => Math.abs(t-y) == x)
  )
yield (x, y, z)
}

forAll (gen_double_1_pos) { _ match {
  case (double_1, double_3, double_2) =>
    val interval = new Interval(double_1, double_2)
    double_3 ~= interval.getSize
  }
}

Figure 3.4: The ScalaCheck generator and property generated by JARVIS from the unit tests in Fig. 3.3.

3.10.1 MATH-1256: Interval bounds

In Apache Commons-Math versions prior to 3.6, the test suite for the Interval class included the code in Fig. 3.3. A bug in the interval class which is not tested in these unit tests was opened as “MATH-1256: Interval class upper and lower check”\(^2\). An Interval object could be created with a lower bound greater than its upper bound, which would result in an invalid interval with a negative size. The bug report shows test code initializing `Interval interval0 = new Interval(0.0, (-1.0));` and showing that it would result in `interval0.getSize()` being `-1.0`.

This bug hinges on the two parameters accepted by the Interval constructor. Since it only requires \(y > x\), it exists in nearly 50% of the parameter space. However, the conventional unit tests in IntervalTest only cover 2 values.

Running JARVIS on IntervalTest.java from release 3.5 yields 9 different scenarios. The scenario testing `getSize` contains two parameterized tests, one for the parameters `double_1`, `double_2` and `double_3`, from the code in `testInterval` and one for `double_1` and `double_2` from the code in `testSinglePoint`. We denote them `pt_3` and `pt_4`, respectively. Since `pt_4 \subseteq pt_3`, the concrete test from `testSinglePoint` is added to the parameterized test for `testInterval`, resulting in \(C^+ = \{(2.3, 5.7, 3.4), (1.0, 1.0, 0.0)\}\).

From the abstraction template library for 3D abstractions, the abstraction selected for these points by the criteria outlined in Section 3.5 is \(|y - z| = x\). JARVIS outputs the code in Fig. 3.4 to generate values matching the abstraction. Running the test with ScalaCheck fails in cases where the upper bound of the interval is negative while the lower bound is generated as always positive. Since the bug exists in nearly 50% of the space, it occurs almost immediately when running the PBT. These cases expose MATH-1256 without the additional unit tests that were later added after it was reported and fixed.

\(^2\)http://issues.apache.org/jira/browse/MATH-1256
3.10.2 MATH-785: Discovering a deep bug with manual intervention

In Commons-Math 3.0, the `FDistributionTest.java` test file included three tests that instantiate an `FDistribution` object, and test that $cdf^{-1}(cdf(x)) = x$. A bug in `ContinuedFraction` caused an exception while computing $cdf^{-1}$ of an `FDistribution` instantiated with sufficiently large values for the numerator and denominator degrees of freedom. This has been reported as “MATH-785: Numerical Underflow in ContinuedFraction”\(^3\). This problem is also related to another bug, MATH-718, and impacted another Commons-Math class, `BinomialDistribution`.

The two concrete tests that appear in the 3.0 version of `FDistributionTest.java` are parameterized into the same parameterized test with parameters $int_1$ and $double_1$, with the data set $C^+ = \{(1, 0.975), (100000, 0.999)\}$. In an environment where precision is not an issue, the ideal abstraction for this case would be $int_1 > 0 \land 0 \leq double_1 \leq 1$. JARVIS selects the abstraction $|0int_1 - double_1| \leq 0.999$. While this is a model example of the difficulty of abstracting from few examples, an important fact—the lack of relation between $int_1$ and $double_1$—has been captured. Values that trigger the bug may be generated from this abstraction, but they are unlikely. Values causing a false positive are far more likely.

Fig. 3.5(a) shows the original code created by JARVIS. We now describe the manual process used to discover the bug:

Running the code produced by JARVIS resulted in the failure: ! Falsified after 0 passed tests.
> ARG_0: (2147483647, -0.5649371160559484). This test failure leads us to two changes in the generator code. The first is to limit $int_1$ to lower values, as when it is sufficiently high, every call to `cumulativeProbability` returns 0 due to lack of precision, and to positive values which are required by the class. The second is to limit $double_1$ to not only values greater than 0 but very near 1 as floating point arithmetic causes the $cdf^{-1}(cdf(x))$ calculation to vary greatly from $x$ through no fault of the code.

Running the code after the changes, seen in Fig. 3.5(b), in ScalaCheck now fails with an exception caused by the bug:
> ! Exception raised on property evaluation.
> ARG_0: (213726, 0.9918989284020788)
> Exception: org.apache.commons.math3.exception.NoBracket-ingException: function values at endpoints do not have different signs, endpoints: [0, 1.001], values: [-0.03, -?]

3.11 Conclusions from JARVIS

Section 3.10.2 shows a prime example of synthesis carrying the brunt of the load, and of multiple biases being applied to select a correct abstraction. However, due to the

\(^3\)http://issues.apache.org/jira/browse/MATH-785
val gen_int_1_pos = for {
  x <- Arbitrary.arbitrary[Int];
  y <- Gen.choose[Double](0.0 * x - 0.999, 0.0 * x + 0.999)
} yield (x, y)
forAll (gen_int_1_pos) { _ match {
  case (int_1: Int, double_2: Double) =>
    val fd = new FDistribution(int_1, int_1)
    val p = fd.cumulativeProbability(double_2)
    val x = fd.inverseCumulativeProbability(p)
    double_2 ~= x
}
}

(a)

val gen_int_1_pos = for {
  x <- Gen.choose[Int](1, 320000);
  y <- Gen.choose[Double](0.975, 0.0 * x + 0.999)
} yield (x, y)
forAll (gen_int_1_pos) { _ match {
  case (int_1: Int, double_2: Double) =>
    val fd = new FDistribution(int_1, int_1)
    val p = fd.cumulativeProbability(double_2)
    val x = fd.inverseCumulativeProbability(p)
    double_2 ~= x
}
}

(b)

Figure 3.5: (a) The ScalaCheck generator and property generated by JARVIS for FDistribution. (b) The code from (a) after manual modifications to the generator in lines 2 and 3.
scarcity of examples, the resulting abstraction, and therefore the resulting synthesized program, are imperfect. Because JARVIS is aimed at a user audience of programmers, it is a fair expectation (as stated in Section 3.7.1) that they review the resulting program and approve it. Because they are programmers, it is also a reasonable expectation that if the resulting test is close to correct, they would prefer to change it rather than to add examples and re-synthesize.

This interaction of the user with code leads us to consider an inherently iterative model of synthesis, one which assumes partial specifications are likely to be insufficient at first try, and harnesses the power of a programmer as the user.
Chapter 4

Programming Not Only by Example: The Granular Interaction Model

When bringing the program into the loop as one of the factors directing the search of the synthesizer, we need to make sure we allow the user to communicate more freely with the synthesizer. In this chapter we present the Granular Interaction Model which allows a more expressive, two-way communication between user and synthesizer than previous models had allowed. The user study performed for this model raises questions about the ability of the user to make correct decisions and what help they need in doing so.

4.1 Introduction

In a development ecosystem where programmers often carry out tasks involving unfamiliar APIs and complex data transformations, program synthesis is both a tool to shorten development times and an aid to small API programming tasks.

Synthesis tools for end-users are available for many purposes, from creating formulae in Microsoft Excel [Gul11] to formulating SQL queries [WCB17]. Tools for expert users who can encode full specifications have also matured enough to be practical [SL08, LNP+12, VYY10].

Expressing Intent   Despite significant progress in synthesis, expressing the user’s intent remains a major challenge. Expert users can write full specifications and express their intent fully [LNP+12, VYY10, BGL+98, IGIS10, Pai90, CGT+16, ISSL+16a, VYB06, VYBR07, HP11], but end-users and programmers trying to solve small tasks often use partial specifications. Partial specifications are available in different forms, depending on the synthesizer: source and target types, input-output pairs, tests, and logical specifications.
Coarse-grained models such as type-driven synthesis present the user with all possible results that satisfy the coarse-grained criteria (e.g., [GKKP13, GRB+14]). This leads to a challenging task: the user must compare a large number of similar programs to select a solution.

Expressing Intent with Examples  A very useful alternative for end-users is to use examples to express intent. Programming by Example (PBE) is a form of program synthesis where the desired behavior is generalized from specific instances of behavior, most often input-output example pairs. This allows an iterative process where, if the synthesized program is not acceptable, additional examples are provided until the target program is reached. This technique is often used either on its own in synthesizers such as [Gul11, WCB17, LWDW01, LH95, WM93, Ant05, WLF15, OSY16, LG14] or as a way to refine the results of type-driven synthesis [OZ15, FCD15].

Insufficiency of Examples for Programmers  PBE is geared towards end-users, but is also useful for more advanced users when the behavior is more difficult to describe than its effect. However, in this interaction model, a user can only do one of two things: accept the program after inspection, or reject it with a differentiating example which will rule it out in the next iteration of synthesis. But some synthesized programs are not all bad: parts of them might be overfitted to the examples, while other parts will be on the right track. Allowing only a full accept or full reject ignores the ability of a programmer to read and understand the program, and to express a more directed, granular feedback, deeming parts of it as desirable or undesirable rather than the program as a whole.

In fact, we hypothesize that, in some cases, it is easier for a programmer to explicitly indicate what is good or bad in a candidate program than to try to express this information implicitly through input-output examples. Moreover, we prove that it is sometimes impossible to express such information through examples.

Programming Not Only by Example  Motivated by the insufficiency of examples, we present a new, granular interaction model that allows a programmer to interact with the synthesizer not only by example but also by providing feedback on parts of the synthesized program. Our interaction model is granular in both directions: from the programmer to the synthesizer and back:

- **Synthesizer → Programmer:** A candidate program is presented together with debug information, showing execution values at different program points. This helps the programmer understand whether the candidate program behaves as expected at intermediate states, instead of relying only on its final output.
- **Programmer → Synthesizer:** A programmer can provide: (i) input-output examples (as in PBE), and (ii) granular feedback on the candidate program by explicitly accepting/rejecting parts of its code.
We tested the granular interaction model by a controlled user-study with 32 developers from both academia and industry. To conduct this study, we developed a synthesizer that interacts with the user in three different ways: holistic (PBE), granular, or both. Our synthesizer also measures interaction times, and records the user-interaction for later analysis.

Our implementation synthesizes functional programs in Scala, a popular functional and object-oriented programming language, used in many big-data processing frameworks (e.g., Spark, Akka). Functional compositions are considered “the Scala way” to approach coding tasks, and so we aim to synthesize them. The same approach also applies to any language that uses functional compositions, which are becoming a standard in modern languages. Notably, such constructs are supported by Java 8 onwards and JavaScript 6 onwards (in JS5 in the popular library lodash).

Advantages of granular interaction The user study strongly supports the hypothesis that it is beneficial to let programmers communicate their understanding of the program explicitly to the synthesizer (by marking parts of it as desirable or undesirable) rather than implicitly (through examples). Several participants in our user study, faced with the inability to rule out an undesired operation in the program using only examples, expressed extreme frustration. We indeed show that this is more common than one would imagine, due to the introduction of redundant or superfluous operations by the synthesizer. As a result, an undesirable operation may be part of several candidate programs along the process, but the holistic PBE model does not allow ruling it out.

We further show that our granular interaction model (GIM) is easier to use, as supported by: (i) a strong preference of participants for granular feedback over examples, and (ii) a significantly shorter iteration time when using granular feedback. It is important to note that granular feedback does not completely replace examples. Participants who were restricted to granular feedback were sometimes forced to use a larger number of iterations, and were more prone to error when accepting the program. We therefore conclude that future synthesizers should integrate both interaction models.

Main Contributions The contributions of this chapter are:

- A synthesis framework with a granular interaction model (GIM) that allows the programmer to approve or reject specific parts of the code of the candidate program rather than just respond to it as a whole and allows a synthesizer to present candidate programs with debug information.
- A theoretical result showing that examples are sometimes insufficient for reaching the desired program. We further show that real PBE sessions exhibit this problem.
- A controlled user study showing that programmers strongly prefer granular feedback instead of examples and can provide granular feedback much faster.
In Section 4.3 we show why examples are not only inconvenient but insufficient to communicate the intent of the programmer. To allow more expressive power, we introduce three additional granular operations in Section 4.4 in addition to examples. In Section 4.4.2, we also introduce debug information for every example provided by the user. Section 4.5 details our experiments on the number of iterations necessary to solve a set of benchmarks with different interaction models. We also present the result of a controlled user study of 32 programmers from academia and industry that shows the benefits of our approach.

4.2 Overview

In this section we provide an overview of our Granular Interaction Model (GIM) for synthesis using a simple example. We start by showing the interaction model of Programming by Example (PBE) and its shortcomings, and then describe how GIM overcomes these shortcomings by using a richer interaction model.

Motivating example Consider the task of writing a program that finds the most frequent character-bigram in a string. Assume that the program is constructed by combining operations from a predefined set we refer to as the vocabulary \( V \). For now, assume that the vocabulary contains standard operations on strings, characters, and lists. In addition, assume that an initial partial specification is provided in the form of an input-output example:

\[
\sigma_0 = "abdfibfcfdebdhgfjgkdfdebd" \mapsto "bd".
\]

In this example, the bigram “bd” is the most frequent (appears 4 times), and is thus the expected output of the synthesized program.

4.2.1 Interaction with a classic PBE synthesizer

Table 4.1 shows the interaction of a programmer with a PBE synthesizer to complete our task. The synthesizer poses a question to the programmer: a candidate program that is consistent with all examples. The programmer provides an answer in the form of an accept, or additional input-output examples to refine the result.

Based on the initial example, the synthesizer offers the candidate program \( q_1 \), which consists of a single method from the vocabulary – takeRight(2), which returns the 2 rightmost characters – applied to the input. The programmer then responds by providing the example \( \sigma_1 \), which is inconsistent with the candidate program, and therefore differentiates it from the target program.

At this point, the synthesizer offers a new candidate program \( q_2 \), which is consistent with both \( \sigma_0 \) and \( \sigma_1 \).
**Task:** find the most frequent bigram in a string

Initial example \((\sigma_0)\) \quad "abdfibfcfdebdfdebdihgfkjfddeb" \(\mapsto\) "bd"

**Question** \(q_1\) \quad **input**

**Problem:** \(\text{takeRight}\) will just take the right of a given string

**Idea:** the frequent bigram needs to be placed in the middle

**Answer** \(\sigma_1\) \quad "cababc" \(\mapsto\) "ab"

**Problem:** this program crops a given input at a constant position

**Idea:** vary the position of the frequent bigram between examples

**Answer** \(\sigma_2\) \quad "bcaaab" \(\mapsto\) "aa"

**Problem:** in all examples the output is the lexicographical minimum of all bigrams in the string (e.g., "aa" \(<\) "bc", "aa" \(<\) "ca", "aa" \(<\) "ab")

**Idea:** have a frequent bigram that is large in lexicographic order

**Answer** \(\sigma_3\) \quad "xyzzzy" \(\mapsto\) "zz"

Table 4.1: The difficulty of finding a differentiating example.

The interaction proceeds in a similar manner. Each additional example may reduce the number of candidate programs (as they are required to satisfy all examples). If the user chooses the examples carefully, the process terminates after a total of 4 examples.

**Finding differentiating examples may be hard** Consider the candidate program \(q_3\). To make progress, the user has to provide an example that differentiates \(q_3\) from the behavior of the desired program. To find a differentiating example, the user must (i) understand the program \(q_3\) and why it is wrong, and (ii) provide input-output examples that overrule \(q_3\), and preferably also similar programs.

By examining the code of \(q_3\), it is easy to see that \(\text{min}\) is a problem: calculating a minimum should not be part of finding a most frequent bigram. Even after understanding the problem, the programmer must still find a differentiating example that rules out \(q_3\). Because the \(\text{min}\) in \(q_3\) takes the (lexicographical) minimum from a list of the bigrams in the input, the programmer comes up with an example where the desired bigram is the largest one, as in \(\sigma_3\).

In this interaction, the programmer had to express the explicit knowledge ("do not use min") implicitly through examples. Coming up with examples that avoid \(\text{min}\)
requires deep understanding of the program, which is then only leveraged implicitly (through examples). Even then, there is no guarantee \texttt{min} will not recur: as we will show in Section 4.3, it cannot be removed completely in this model. In this case, since the programmer already knows that programs using \texttt{min} should be avoided, this information is best communicated this information explicitly to the synthesizer.

4.2.2 Interaction through a granular interaction model

GIM improves PBE by employing a richer, \textit{granular} interaction model. On the one hand, the synthesizer supplements the candidate programs by debug information that assists the programmer in understanding the programs and identifying their good and bad parts. On the other hand, the user is not restricted to providing semantic input-output examples, but can also mark parts of the program code itself as parts that must or must not appear in any future candidate program. This allows the user to provide explicit, syntactic, feedback on the program code, which is more expressive, and in some cases allows the synthesizer to more aggressively prune the search space.

The GIM interaction for the same task of finding the most frequent bigram is demonstrated in Table 4.2. Question 1 is as before: the synthesizer produces the candidate program \texttt{input.takeRight(2)}. In contrast to classic PBE, the granular interaction model provides additional debug information to the user, showing intermediate values of the program on the examples. This is shown as comments next to the lines of the synthesized program. For \texttt{q1}, this is just the input and output values of the initial example. In the next steps this information will be far more valuable.

Given \texttt{q1}, the programmer responds by providing \textit{granular feedback}. Using GIM it is possible to narrow the space of programs using syntactic operations. Presented with \texttt{input.takeRight(2)}, the user can exclude a sequence of operations from the vocabulary, in this instance \texttt{takeRight(2)}, ruling out \textit{any program} where \texttt{takeRight(2)} appears. This also significantly reduces the space of candidate programs considered by the synthesizer.

The synthesizer responds with \texttt{q2}. Note that in such cases the debug information assists the programmer in understanding the program, determining whether it is correct, or, as in this case, identifying why it is incorrect. To rule out \texttt{q2}, the user rules out the sequence \texttt{drop(1).take(2)}, as the debug information shows the effect (“take the second and third character of the string”), and the user deems it undesirable at any point in the computation to truncate the string, as all characters should be considered.

The synthesizer responds with \texttt{q3}. This candidate program contains something the programmer would like to preserve: the debug information shows that the prefix \texttt{input.zip(input.tail)} creates all bigrams in the string. The user can mark this prefix to \textit{affix}, or to make sure all candidate programs displayed from now on begin with this prefix. This removes all programs that start with any other function in \( \mathcal{V} \), effectively slicing the size of the search space by \( |\mathcal{V}| \). Another option (multiple operations
stemming from the same program are not only allowed but encouraged) is to exclude \texttt{take(2)} since the resulting truncation of the list is undesirable.

Eventually, the synthesizer produces the following program:

\begin{verbatim}
input //"abdfibfcfdebdfdebdbihgfjkjfdedb"
.zip(input . tail) // List((a, b), (b, d), (d, f), (f, i), (i, b), (b, f), ...)
.map(p => p._1.toString + p._2) // List("ab", "bd", "df", "fi", ...)
.groupBy(x => x)//Map("bf"->List("bf"), "ib"->List("ib"), ...)
.map(kv => kv._1 -> kv._2.length)//Map("bf"->1,"ib"->1,...)
.maxBy(_._2) //_//"bd"

which does not discard any bigram, counts the number of occurrences, and retrieves the maximum. This program is accepted.

Below we summarize the key aspects of GIM, as demonstrated by the above example.

**Key Aspects**

- **Granular feedback:** the programmer can provide feedback (keep/discard) on parts of the program, in addition to input-output examples. The ability to give explicit feedback on the code itself provides an alternative (and complementary) way to interact with the system without crafting potentially complicated differentiating examples.

- **Assisting the User with Debug Information:** the synthesizer provides debug information on intermediate states of the program in order to assist the user. Candidate programs are supplemented with debug information that helps the programmer understand the good and bad parts of a candidate program.

- **Insufficiency of Examples:** examples are both inconvenient and insufficient to communicate a programmer’s intent. Other operations are needed to allow a programmer to filter programs not only according to semantic equivalence but also according to additional criteria such as readability, best practices and performance.

**4.3 The Insufficiency of Examples**

In this section, we show the importance of extending the user’s answer model beyond input-output examples. We examine more formally the scenario described in Section 4.2.1, where the user has seen an undesirable program component and would like to exclude it specifically. We will show that this is not always possible, i.e., that examples are insufficient to communicate the user’s intent.

As seen in Section 4.2.1, the user wishes to rule out the function \texttt{min}, but simply providing an example to rule out the current program might not be enough to remove
min from all candidates to ensure it never recurs. We now formally prove it is impossible to completely remove methods like min from the search space using examples.

We recall the definition of equivalence between programs. Programs \( m_1 \) and \( m_2 \) are equivalent if \( \langle m_1 \rangle = \langle m_2 \rangle \). We use this to prove the following claim:

**Claim 4.3.1.** Let \( v \in \mathcal{V} \) be a letter such that there exists a program \( m \) that is equivalent to \( m^* \) and contains \( v \). Then examples alone cannot rule out the letter \( v \in \mathcal{V} \) from candidate programs.

The proof follows since examples can only distinguish between programs that compute different functions.

Next we show that Claim 4.3.1 is applicable to methods that are prevalent in programming languages and extremely useful in some contexts, and therefore are likely to find their way into the vocabularies used in synthesis. We focus on redundant code, and specifically on two methods to create redundant code in a program: invertible methods and nullipotent methods.

**Invertible methods** are methods with an inverse that, when applied in sequence lead back to the initial input. For instance, reverse on a list is invertible and its own inverse, as \( \text{in.reverse.reverse} \) will be identical to \( \text{in} \). An invertible method can always be added to the target program along with its inverse, resulting in an equivalent program.

**Nullipotent methods** are methods that, when applied, lead to the same result as not being applied. While this is often context-sensitive, e.g. calling toList on a list or mkString on a string, there are calls that will always be nullipotent, such as takeWhile(true). Because some methods are nullipotent only in a certain context, they may be in a synthesizer’s vocabulary, and end up in the program space in contexts where they are nullipotent. It is easy to construct a program that contains nullipotent methods and is equivalent to the target program. Hence, similarly to invertible methods, these methods cannot be eliminated by examples.

If we examine the programs in Table 4.1, we notice the user trying to rid themselves of a component from \( \mathcal{V} \), the call to min. The target program of the synthesis session is the Scala program

```scala
input.zip(input.tail).map(p => p._1.toString + p._2)
groupBy(x => x).map(kv => kv._1 -> kv._2.length)
.maxBy(._2)._1
```

Let us now construct an equivalent program by appending to it an invertible pair of functions in sequence: sliding(2).min. The function sliding(2), when applied to a string of length 2 will return List("dc"), and min when applied to list of size 1 will return the only member of the list. This means there is a program that is equivalent to
on every input, and contains \( \min \). As such, given any number of examples applied \( \min \), a letter from \( \mathcal{V} \), will not be ruled out entirely.

This construction is possible for many target programs, under many vocabularies, showing that it is often impossible to discard an undesirable member of the alphabet or an undesired sequence using examples alone.

Furthermore, since many existing PBE synthesizers prune very aggressively based on observational equivalence, or equivalence based only on the given examples, programs that do not include the undesired component may not be available anymore as they have been removed from the space.

It’s easy to consider synthesis from a purely functional standpoint, and see the iterative PBE process as sufficient: every time a program \( m \) that is returned from the synthesizer is not functionally correct, there exists an example \((\iota, \omega)\), and by extension a set of examples \( I \) which will separate \( m \) and \( m^* \) into two equivalence classes.

Every time the user provides a new example \((\iota, \omega)\) that rules out the current candidate program \( m \), or in other words \([m](\iota) \neq \omega\), this creates a refinement, \( (\equiv \cup \{\iota\}) \subseteq (\equiv j) \), that breaks up at least one observational equivalence class—that from which the candidate program was taken: the program \( m \) does not uphold \((\iota, \omega)\) and will be part of one equivalence class, while \( m^* \), the target program, does uphold \((\iota, \omega)\) and will be part of another. If the equivalence class of \( m^* \) in this partition contains undesirable programs, the partition is simply not refined enough.

These properties leave us with the need to define a more expressive, granular model.

The practical implications of claim 4.3.1 are discussed in Section 4.5.4, which examines the existence of method sequences deemed undesirable by users in candidate programs. The data as well as opinions collected from users show that the inability to remove an undesirable letter from the alphabet has real-world consequences, which add to the user’s frustration with the synthesizer (see Table 4.6).

### 4.4 The Granular Interaction Model

In this section, we describe the Granular Interaction Model (GIM), which extends the PBE model with additional predicates. Namely, predicates in GIM include examples, but also additional predicates. The key idea is to add a broader form of feedback from the user to the synthesizer than has been available in PBE. We begin by describing the operations and the type of feedback that each such predicate allows the user to provide the synthesizer with, and discuss the observed uses of each.

#### 4.4.1 Granular predicates

In the setting of functional compositions, we present GIM with three syntactic predicates. We refer to these predicates as granular since they impose constraints on parts of the program rather than on its full behavior, as captured by the function it computes or its
input and output types. We will also discuss other, possible predicates.

Given a candidate program $m = f_n(f_{n-1}(\ldots \text{input} \ldots))$, we introduce the following predicates, to be tested against other programs $m' = f'_m(f'_{m-1}(\ldots \text{input} \ldots))$:

1. **remove**$(f_i, \ldots, f_j)$ where $i \leq j$: will hold only for programs $m'$ where $\neg \exists k. f'_k = f_i \land \cdots \land f'_{k+i-j} = f_j$

2. **retain**$(f_i, \ldots, f_j)$ where $i \leq j$: will hold only for programs $m'$ where $\exists k. f'_k = f_i \land \cdots \land f'_{k+i-j} = f_j$

3. **affix**$(f_0, \ldots, f_i)$: will hold only for programs $m'$ where $\forall j \leq i. f_j = f'_j$

The **remove** operation rules out a sequence of one or more method calls as undesirable. For the example in Section 4.2, to rule out min the user would simply add the predicate **remove**(min). However, should the user rule out a sequence longer than a single method, this would apply to the sequence as a whole: using the predicate **remove**(reverse, reverse) does not exclude the reverse method, only two consecutive invocations of it that cancel out.

The **retain** operation defines a sequence that must appear in the target program. It is similarly defined for sequences: when applied to a single method, it forces the method, and when applied to a sequence it forces the sequence, in-order. It can be viewed as creating a procedure and then deeming it as desirable.

However, since **retain** is not dependent on the location of the procedure in the program, we add an additional predicate for not only setting a procedure but forcing its location to the beginning of the program. The **affix** predicate will essentially narrow the search space to sub-programs that come after the desired prefix.

**Additional predicates**  As these three operations are highly expressive and easy to understand, we have focused our experiments on them, but they are by no means the only possible predicates. Many other granular operations exist. For instance, the user can reason about intermediate states of the program by demanding or excluding certain intermediate states for a given input. A user can also require an error, or an error of a certain kind, for a given input. Section 4.4.4 will expand on the reasons to select certain expansions to the interaction model over others.

### 4.4.2 Adding a debugging view of the code

GIM assumes an interaction with users who are comfortable reading code. This means not only that more can be expected from them, but that they can be assisted in ways currently not offered by synthesizers. Just as the interaction from the user to the synthesizer can be granulated, so can the interaction from the synthesizer to the user.

PBE tools like FlashFill only show the user the output of running the program on an input. Other tools that do show code show the program while guaranteeing that it
satisfies all examples in $E$. In a functional concatenation, it is possible to show the user the result of each subprogram, on each $e \in E$. This means that even for some unfamiliar $f \in V$, the user can still gauge its effect and determine by example whether that effect is desired.

**Example 4.4.1.** Let us consider the case where input is a list of strings, and the user is presented with the candidate program `input.sliding(3).map(l => l.mkString)`. While familiar with the `mkString` method, which formats a list into a string, and with mapping a list, the user has never encountered `sliding`.

The user could look up the method and read up on its behavior. However, oftentimes its behavior will be simple enough to understand by its operation within the program. Consider, for example, the following intermediate program states:

```scala
input // List ("aa", "bb", "cc", "dd", "ee")
.sliding(3) // List (List ("aa", "bb", "cc"), List ("bb", "cc", "dd"), ...)
.map(s => s.mkString) // List ("aabbcc", "bbccdd", "ccddeee")
```

Provided with these states, the user can understand that `sliding` returns a list of sublists of length $n$ beginning at each position in the list – a sliding window of size $n$.

### 4.4.3 Enabling the User

Having introduced the formal framework for predicates, we now wish to leverage it to create a user interaction model. We suggest the following iterative process, which we have implemented for the user study in Section 4.5.3.

A candidate program is displayed to the user alongside the debug information. The top image in fig. 4.1 shows this in our UI. The user is now able to study the program and accept or reject it.

The goal is to allow a user who is dissatisfied with the program to directly express the source of dissatisfaction as easily as possible, using predicates. Towards this end, we let the user point out a portion of the program (e.g. by right-clicking it) and mark it as desirable or undesirable, as seen in the bottom image in fig. 4.1.

This process of easily providing feedback on the program turns predicates into a convenient tool for feedback to the synthesizer.

### 4.4.4 Enabling the synthesizer

As we have seen, the choice of predicates is crucial from the user’s perspective. However, it is also important for the synthesizer to be able to use them in maintaining and updating a representation of the search space. To complete this section, we show how the predicates described in Section 4.4.1 are naturally utilized by a synthesizer for the domain of linear functional concatenations.
Enumerating synthesizer  The state of the art in program synthesis hinges on enumerating the program space in a bottom-up fashion \cite{ARU17,FCD15,AFSSL16}. For the domain considered in this chapter, bottom-up enumeration consists of concatenating method calls to prefixes already enumerated, starting with the program of length 0, `input`. This enumeration is restricted by types, i.e., by compilation. The search space in this synthesizer can be represented as an edge-labeled tree where the root is the program `input` and each edge is labeled by a method name from \( V \). Each finite-length path in the tree represents the program that is the concatenation of every label along the path. The tree is initially pruned by compilation errors (i.e., if \( f \in V \) does not exist for the return type of \( m \), it will be pruned from the children of the node representing \( m \)). It now represents the candidate program space for an unconstrained synthesizer state.

We can see that every program deemed undesirable by the operations `affix` and `remove` cannot be extended into a desirable program. Therefore these extensions can be discarded and the tree representing the candidate space can be pruned at the nodes of these programs.

This is an example of predicates that are well suited to the representation of the state of the synthesizer, in that they not only aid the user but also help guide the search of the space. Since the combination of the enumeration and these predicates is monotone, a program that was pruned from the search space will never have to be considered in a future, more constrained iteration. This means that the synthesizer does not need to be restarted across iterations. However, even if it is, these predicates will allow it to construct a much smaller search space to begin with.

### 4.5 Evaluation

To evaluate our approach, we compared three interaction models:

1. PBE: replicating the state of the art in synthesis, the user can communicate with the synthesizer via input-output pairs.
2. Syntax: testing the new operation set proposed in Section 4.4, the user can communicate with the synthesizer via syntactic predicates on the program.

3. GIM: testing the full model, the user can communicate via both sets of predicates. We limited the test of the granular interaction model to three operations that are relevant to functional compositions and are easy to understand. Therefore, we selected the operations detailed in Section 4.4.1 as our basic set of granular operations.

We conducted two studies:

1. A study of ideal sessions with different operations (i.e., families of predicates) for a set of benchmarks.
2. A controlled user study which tests the usability of a GIM synthesizer for programmers and the benefits when measured against a control group using PBE.

**Synthesizer** We implemented a simple enumerating synthesizer described in Section 4.4.4 in Scala, using the nsc interpreter (used to implement the Scala REPL). The vocabulary $\mathcal{V}$ is provided to the algorithm, and programs are compiled and evaluated on the inputs.

In order to support the study in Section 4.5.2, the synthesizer accepts input of additional examples, rejection of the current program, or of affix, remove and retain predicates. In order to support the user study in Section 4.5.3, it also precomputes the space of valid programs.

4.5.1 Problem set

We conducted the studies using a set of functional programming exercises from three different domains: strings, lists and streams. The exercises were collected from Scala tutorial sites and examples for using MapReduce. The tasks, described in Table 4.3, were each paired with a vocabulary and an initial set of examples.

**Discussion** As seen in Table 4.3, the set of valid programs is significantly smaller than $|\mathcal{V}|^{|m^*|}$, but in many cases the space still contains thousands or tens of thousands of programs. There is also a fair amount of inherent ambiguity over the initial example set $\mathcal{E}_{init}$, as can be seen in the “reject only” column, representing the set of all programs up to length $|m^*|$ that match $\mathcal{E}_{init}$. This means that, even when limiting the search space to the known length of the target program, we would start with hundreds or thousands of matching programs that need to be filtered by the user.

4.5.2 Ideal synthesis sessions

**Experimental questions** For each task in the problem set we answered the following: under the ideal conditions of an expert user and knowledge of the target program, how many questions (i.e., candidate programs) are posed to the user for each predicate family?
Test setup  In order to answer these questions, each task in the problem set was run in four settings:

- **Reject only**: no operations except rejecting the current program. This essentially enumerates programs that match the initial example set.
- **PBE, Syntax, and GIM**: as described above, all with the addition of a reject operation.

Examples and other predicates were selected by an expert user (author of this thesis), making an effort to create a run with fewer iterations and more aggressive pruning of the space in each iteration.

Results  Table 4.3 shows the results for each of the programming tasks. As can be seen from the table, in ideal (i.e. thoroughly optimized, expert user) runs, the number of questions produced by the synthesizer for a PBE run was lowest. This was not unexpected: carefully selected examples are a fast way to differentiate between programs. Examples selected in less ideal conditions are left to the following section. But we also see that in a run allowing all predicates, substantially fewer questions were asked than when using syntactic predicates only, with no more than one example.

The synthesizer and its outputs are available at http://bitbucket.org/hilap/scala-enumerating-synthesizer/.

4.5.3 User study  
To test the interaction between programmer and synthesizer, we conducted a user study, where we compared the interaction of programmers with the synthesizer using the three families of operations: **PBE** (control), **Syntax**, and **GIM**.

Research questions  We examined the following questions:

1. **Are answers consisting of syntactic predicates easier or faster to generate than example predicates?** This question was examined first by comparing, for each task, the average and median iteration times with the synthesizer for the control group (PBE) and the Syntax group. Second, when users were allowed both (GIM), the time spent on iterations where they provided examples was measured against their average time.

2. **Is the total time to solution improved by adding or exchanging the available predicates?**

3. **Are users able to reach a correct program using each of the predicate sets?**

4. **Do users prefer examples?** This question examined the choices made by the participants in the GIM group, who could choose between all predicates. We tested how often examples were chosen, and whether the task being solved affected this preference.

5. **Are users in PBE sessions distracted by undesirable sequences that cannot be removed?** We tested PBE sessions for recurrence of sequences deemed undesirable by
users in the Syntax and GIM groups, to try to determine whether these recurred enough to distract users. We also checked for acceptance of equivalent programs with superfluous elements as mentioned in claim 4.3.1. Anecdotal opinions offered by participants are also presented.

Most questions were examined on all participants. We show data for the small set of users experienced in Scala against those new to Scala when the difference is of interest.

**Test setup** 32 developers participated in the study. They consist of 7 undergraduates in their final year of a CS degree, 9 graduate students in CS, most with a history as developers outside academia, and 16 industry developers employed by four different companies. Of the 32, 8 had prior experience with the Scala programming language.

The participants in the study were evenly distributed between three test groups: PBE, Syntax and GIM. Each participant was randomly assigned to one of the test groups. Not all participants performed all tasks (scheduling constraints were cited for the most part). The order of the tasks was randomized for each user.

The reject operation was not allowed in any group, forcing users to provide the process with new information as they would in any state of the art synthesizer, rather than just iterate the program space.

Each participant was asked to use the synthesizer to solve three programming questions. The three problems—frequword, nonemptylines, and histogram—were selected from the tasks tested in section 4.5.2 because of their high level of ambiguity based on the initial example, and their requiring no additional libraries or definitions outside the Scala standard library to solve (i.e., the programs could be run in a Scala console with no imports or definitions).

Participants were given a short introduction to Scala, if they were not already familiar with it, and assisted themselves with a Scala REPL, but no online sources or documentation.

Correctness of a participant’s solution was defined functionally, using predetermined tests, and including no nullipotent calls (these were “equivalent”). There might be several correct programs in the space; e.g., when counting non-blank lines, solutions allowing for CRLF line-ends were accepted as long as they correctly handled LF.

**Implementation** Participants performed the tasks using the UI shown in Figure 4.1. The space of programs was precomputed by the enumerating synthesizer from Section 4.5.1 and over the same initial inputs, up to a program length of 6. In each iteration a program that upholds all predicates given by the user was selected from the set of programs. Selection used a hash-based criterion to prevent lexicographical ordering and favoring of short programs, in order to also show the user complex programs. At the end of every iteration the user’s answers were added to the synthesizer’s state and the programs are filtered accordingly. If the precomputed set was exhausted, the user was given the option of starting over or abandoning the current task.
4.5.4 User study results

We address each question individually.

**Question 1: Are answers consisting of syntactic predicates easier or faster to generate than example predicates?**  The average and median times per iteration are shown in Table 4.4. Medians are also shown in Fig. 4.2.

We examined the distributions of data using the Mann-Whitney test. The threshold for statistical significance was chosen as \( p < 0.05 \). A significant difference was found in the time per iteration between the control (PBE) group and the syntax-only group for all tests: histogram (131.97s, 59.97s, \( p = 0.03 \)), nonemptylines (168.17s, 60.56s, \( p = 0.03 \)) and freqword (71.27s, 50.34s, \( p = 0.04 \)). A significant difference was found between the control group and the GIM group for two of the three tests: histogram (131.97s, 96.18s, \( p = 0.03 \)), nonemptylines (168.17s, 65.78 s, \( p = 0.03 \)), but not for freqword (71.27s, 53.84s, \( p = 0.058 \)). Additionally, a significant difference was found between the syntax-only group and the GIM group for one test: histogram (59.97s, 96.18s, \( p = 0.037 \)), but not for nonemptylines (60.56s, 65.78s, \( p = 0.33 \)) or freqword (50.34s, 53.84s, \( p = 0.19 \)).

These results imply that, with the exception of the freqword test for the GIM group, iteration time is faster when using either syntax only or both syntax and example predicates than when solving the same problem using PBE alone. Additionally, with the exception of the histogram task, the slowdown in iteration time between syntax-only and GIM seems to be coincidental.

In addition, we looked only at the session for users in the GIM group and within each session examined the time to create an example against the average iteration time. There is a slowdown of 19.5% in iteration time with an example, and we see that this difference is statistically significant (75.03s, 90.11s, \( p = 0.049 \)).

We can therefore answer question 1 in the affirmative on both counts: syntactic predicates are faster to generate than examples, both when examining the test groups against the PBE group, and when examining the users with access to both against themselves.

**Question 2: Is the total time to solution improved by adding or exchanging the available predicates?**  We noticed a change in the median total time between the control (PBE) and the other groups (Syntax and GIM), indicating a possible slowdown. However, this change was not statistically significant for any of the individual tests or for the unification of all tests (\( p > 0.25 \) for all). Therefore, while we do not answer question 2 in the affirmative – as the total time was not improved in either of the test groups – we can also say that the evidence of a slowdown may be coincidental.

**Question 3: Are users able to reach a correct program using each of the predicate sets?**  The correctness results in Table 4.4 are visualized in Fig. 4.4. Aside
from the histogram task, completed by all users, all other tasks had some users stopping without accepting a program. The success percentage in reaching any functionally correct response is highest for PBE (100%, 73%, 90%), lowest for Syntax (78%, 57%, 87%), and rebounds with GIM (90%, 77%, 80%) to levels close to the control, even overtaking it for the nonemptylines task.

**Question 4: Do users prefer examples?** A summary of how often users chose examples appears in Table 4.5 and Fig. 4.3. We can see a distinction between users familiar with Scala and users who are not. While users familiar with Scala used examples in every task, users unfamiliar with Scala did not: in every task, at least one user—and as many as 1/3 of the users—avoided them altogether. The proportional part of examples out of the total predicates used in the task is fairly low for the entire test group, ranging from 31% to 37.5% (median).

We compared users familiar and unfamiliar with Scala and found that the preference for examples is inverse between the two groups: users familiar with Scala selected far more examples (all over 60% examples) for the histogram task and preferred other predicates for the frequword task. Conversely, those unfamiliar with Scala preferred examples (but not as overwhelmingly, half the participants using over 30% examples) for frequword and favored other predicates (half the participants using under 20% examples) for histogram. This seems to suggest a relationship with the difficulty of the task—histogram is harder to solve than frequword. Despite this, even when examples were favored, they were never the only tool used.

**Question 5: Are users in PBE sessions distracted by undesirable sequences that cannot be removed?** We first identified such sequences by counting how many users who could remove them (i.e., had access to a remove predicate) actually did so, then tested sessions of users from the PBE group for their appearance. Table 4.6 shows the results. It is important to note that not all commonly removed sequences appeared in PBE sessions, itself an indicator of the extent to which syntax operations change the traversal of the search space.

These undesirable sequences appeared up to 7 times in a single user session. Some of these sequences distracted (i.e. kept reappearing) up to 2/3 of the users performing a task, and on average 22.2% of the users. Furthermore, a distracting sequence appeared, on average, about 3 times in each session. This shows that the inability to remove a letter or sequence discussed in claim 4.3.1 is neither a purely theoretical problem, nor a problem leading only to equivalent rather than correct programs, as seen in Table 4.4, but a real distraction from the ability to synthesize over an expressive vocabulary.

Figure 4.4 also shows that in two of the tasks (histogram and nonemptylines) most or all PBE users ended up accepting a program with superfluous elements. For example, in many histogram sessions a program was accepted with a call of toMap on a map, and in many nonemptylines sessions a program was accepted that called filterNot(c
Figure 4.2: Median iteration time per task in each test group. Significant change from PBE is indicated by *.

=> c == 'r' || c == 'n') on a list of strings. Both are nullipotent elements: toMap creates a map from a map, and filterNot(c => c == 'r' || c == 'n') compares strings to characters and so always filters nothing.

In addition, when PBE users stopped at an equivalent program rather than the target program, we tested the number of iterations spent in the same equivalence class (i.e. presented with the same candidate program) before accepting the program. While most users accepted equivalent programs immediately, one user performing the histogram task tried an additional iteration and one user tried two additional iterations. For nonemptylines, two users tried an additional iteration and one tried two additional iteration. Altogether, in 22% of the sessions users tried unsuccessfully to improve upon the program they already had, either trying to get rid of a nullipotent element or not realizing it has no influence, before finally accepting it.

We chose not to tackle the questions of user preference and measures of distraction with a questionnaire, sticking instead only to empirical results. Despite that, we wish to recount several anecdotes from the course of the experiment that may help shed light on the behavior observed. Users in the PBE test group expressed very specific frustration on several occasions such as “it insists on using take(5) no matter what I do” while solving the frequword task, or “I couldn’t get rid of these nonsense functions, I just wanted to shake it” after solving the nonemptylines task.

The user study UI and recorded user sessions are available at http://bitbucket.org/hilap/gim-ui/.

4.5.5 Discussion and conclusions

In this section we discuss the results of the study.

**Speed and ease of use** We see a speedup of iteration time when using examples versus other predicates. The change is greatest between the PBE and Syntax groups, with a smaller speedup when examining the GIM group against itself. We may attribute
We conclude that when combined with a low preference for examples, syntax predicates are easier for the user in general.

In addition, considering the shorter iteration time, the increased number of iterations (itself statistically significant) and the lack of significance in the change in total time, we conclude that changing the predicates does not change time spent on synthesis tasks. It simply leads to using more, but shorter and easier, iterations.

**Distracting elements and user frustration** Much of the frustration users in the PBE group expressed had to do with recurring program elements they thought were useless. Recurring undesirable sequences were experienced by up to two-thirds of
the users and recurred on average 3 times during the session, certainly explains their frustration. In addition, some PBE users wasted time and effort trying to remove elements that cannot be removed. We therefore conclude that avoiding this distraction by giving users more tools would, at the very least, make for more content users.

**Helpfulness of debug information** We attribute the success rate and relatively short use times of a set of developers who have never seen Scala to the guidance offered by debug information. We did not target this specifically in the experiment, but 9 separate users told us that it was anywhere from “helpful” to “lifesaving” in understanding unfamiliar methods and keeping track of examples.

**Correctness with syntax operations** The Syntax group reached fewer functionally correct programs than the PBE group, and the GIM group did almost as well as the PBE group. We attribute this to the helpfulness of debug information: it seems to be easier to make a correct decision about a program when presented with its breakdown over additional examples, rather than just the single initial example available to the Syntax group.

**Preferred operations** When considering all users in the GIM group, there appears to be a very strong preference for syntactic predicates over examples for all tasks. However, for a sub-group, preferences may be reversed: Users familiar with Scala preferred more examples than those unfamiliar with Scala, and preferred examples over other predicates in the harder task, histogram, and predicates over examples in the easier task, frequword. This may have to do with their ability to better understand candidate programs: savvier programmers read the programs more easily and so prefer to break the observed behavior with examples, while inexperienced programmers focus on individual program elements. This remains a conjecture as there were only 2 users familiar with Scala in the GIM group.

### 4.6 Threats to Validity

**Cross validation** The study was not cross-validated (i.e., having each user perform tasks in several groups). Because the predicate families include each other, we felt it would create a bias toward some operations based on order. As cross validation should not be used when it creates bias, we decided against it. We tried to negate some of the differences between individual programmers by drawing participants from similar backgrounds – same year in university, developers in the same department – and then dividing them evenly.

**Drop-outs** Two of the 32 participants did not complete all tasks, both from the Syntax group. This introduces two problems: first, since the users in question only
performed one or two sessions, these are sessions where they are still less adept. Second, they create an imbalance in the test groups when comparing the exercises after they dropped out. However, since the impacted group was the Syntax group, most conclusions about the GIM test group are unaffected.

**Sampling of population** An external validity issue mentioned in Section 4.5.5 is the relatively small percentage of participants familiar with Scala – only 25% of the participants in the study, and as small as 20% in some groups due to random assignment. As mentioned, this prevents us from making general claims about differences between programmers based on their familiarity with Scala. However, we can still generalize our claims with regard to programmers working in a language they have not encountered before – the majority of the participants. In addition, our sample of undergraduates is not random but consists of students who felt familiar enough with functional programming to agree to participate. This may skew the ability to generalize. We hope this will not significantly affect the compiled results as undergraduates comprise less than 22% of the participants.
**Task:** find the most frequent bigram in a string

**Initial specifications**

```
"abdfibfcfdebdhgfjkjdfjkebd"\rightarrow"bd"
```

**Question**

```
input //"abdfibfcfdebdhgfjkjdfjkebd"
takeRight(2)//bd
```

**Problem:** `takeRight` will just take the right of a given string

**Idea:** `takeRight` will never be useful since we always want to consider every element. Remove `takeRight` from the result program.

**Answer**

```
Remove(takeRight(2))
```

**Question**

```
input //"abdfibfcfdebdhgfjkjdfjkebd"
drop(1)//"bdfjfcfrdfdebdhgfjkjdfjkebd"
take(2) //"bd"
```

**Problem:** this program crops a given input at a constant position

**Idea:** we don’t want to crop anything out, so these functions have no place in the result program.

**Answer**

```
Remove(drop(1).take(2))
```

**Question**

```
input //"abdfibfcfdebdhgfjkjdfjkebd"
zip(input.tail) //List((a,b),(b,d),(d,f),...)
take(2) //List((a,b),(b,d))
map(p => p._1.toString + p._2) //List ("ab","bd")
max //"bd"
```

**Problem:** while the beginning of this program is actually good (dividing the program into bigrams) and so is the mapping of a 2-tuple to a string, `take(2)` truncates the bigram list.

**Idea:** preserve what is good in the program and remove `take(2)` on its own and not just as part of a sequence.

**Answer**

```
Affix(zip(input.tail))
Remove(take(2))
Retain(map(p => p._1.toString + p._2))
```

**Table 4.2:** Providing granular, syntactic feedback.
### Table 4.3: The test setup of 14 synthesis experiments, showing the ambiguity inherent in $E_{\text{init}}$, and the number of iterations to the target program in an ideal synthesis session with each available set of operations. Parentheses indicate examples used.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>candidate reject</th>
<th>Number of Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>droppathletter</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>freqbigram</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>freqword</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>linesinfile</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>nonemptylines</td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 4.4: Summary of the three tasks performed in the user study (all users).

<table>
<thead>
<tr>
<th>task</th>
<th>no. of sessions</th>
<th>iteration group</th>
<th>number of time (sec)</th>
<th>iterations</th>
<th>correct target</th>
<th>equiv. answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td></td>
<td>avg</td>
<td>med</td>
<td>avg</td>
<td>med</td>
<td>finished</td>
</tr>
<tr>
<td>histogram</td>
<td>PBE</td>
<td>11</td>
<td>163.34</td>
<td>131.97</td>
<td>2.45</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>Syntax</td>
<td>9</td>
<td>86.27</td>
<td>59.97</td>
<td>12.11</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>GIM</td>
<td>10</td>
<td>98.78</td>
<td>96.18</td>
<td>8.90</td>
<td>7.5</td>
</tr>
<tr>
<td>no. lines</td>
<td>PBE</td>
<td>11</td>
<td>170.13</td>
<td>168.17</td>
<td>2.73</td>
<td>2.0</td>
</tr>
<tr>
<td>with text</td>
<td>Syntax</td>
<td>8</td>
<td>82.16</td>
<td>60.56</td>
<td>10.50</td>
<td>9.5</td>
</tr>
<tr>
<td></td>
<td>GIM</td>
<td>11</td>
<td>78.26</td>
<td>65.78</td>
<td>8.82</td>
<td>8.0</td>
</tr>
<tr>
<td>most</td>
<td>PBE</td>
<td>11</td>
<td>114.52</td>
<td>71.27</td>
<td>4.45</td>
<td>4.0</td>
</tr>
<tr>
<td>frequent</td>
<td>Syntax</td>
<td>10</td>
<td>58.28</td>
<td>50.34</td>
<td>22.10</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>GIM</td>
<td>11</td>
<td>79.87</td>
<td>53.84</td>
<td>8.82</td>
<td>8.0</td>
</tr>
</tbody>
</table>

Table 4.4: Summary of the three tasks performed in the user study (all users).
<table>
<thead>
<tr>
<th></th>
<th>no. of sessions used</th>
<th>percent examples per user</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>med</td>
</tr>
<tr>
<td>all users</td>
<td></td>
<td></td>
</tr>
<tr>
<td>histogram</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>nonemptylines</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>frequword</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>users</td>
<td></td>
<td></td>
</tr>
<tr>
<td>histogram</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>nonemptylines</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>frequword</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>familiar users</td>
<td></td>
<td></td>
</tr>
<tr>
<td>histogram</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>nonemptylines</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>frequword</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>unfamiliar users</td>
<td></td>
<td></td>
</tr>
<tr>
<td>histogram</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>nonemptylines</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>frequword</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.5: Proportional part (%) of examples in the predicates provided by GIM group users. Some used no examples at all, none used only examples.

<table>
<thead>
<tr>
<th>removed sequence</th>
<th>no. of lines</th>
<th>PBE</th>
<th>GIM/Syntax times seen</th>
<th>distracting occurrences</th>
<th>users</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tail</td>
<td>84.2% (16)</td>
<td>1</td>
<td>4</td>
<td>2.8</td>
<td>45.5% (5)</td>
</tr>
<tr>
<td>takeWhile(c =&gt; c != &quot;\n&quot;)</td>
<td>73.7% (14)</td>
<td>1</td>
<td>4</td>
<td>2.7</td>
<td>54.5% (6)</td>
</tr>
<tr>
<td>filterNot(c =&gt; c=='\r'</td>
<td></td>
<td>c=='\n')</td>
<td>57.9% (11)</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>filter(!l.isEmpty)</td>
<td>27.3% (3)</td>
<td>0</td>
<td>3</td>
<td>2.3</td>
<td>27.3% (3)</td>
</tr>
<tr>
<td>tail.takeWhile(c =&gt; c != &quot;\n&quot;)</td>
<td>15.8% (3)</td>
<td>1</td>
<td>4</td>
<td>2.8</td>
<td>45.5% (5)</td>
</tr>
<tr>
<td>takeRight(1)</td>
<td>100.0% (12)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
<tr>
<td>drop(10)</td>
<td>84.6% (11)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
<tr>
<td>drop(1)</td>
<td>76.5% (13)</td>
<td>0</td>
<td>3</td>
<td>2.5</td>
<td>16.7% (2)</td>
</tr>
<tr>
<td>takeRight(6)</td>
<td>76.2% (16)</td>
<td>1</td>
<td>3</td>
<td>2.4</td>
<td>58.3% (7)</td>
</tr>
<tr>
<td>dropRight(1)</td>
<td>71.4% (15)</td>
<td>0</td>
<td>7</td>
<td>3.3</td>
<td>33.3% (4)</td>
</tr>
<tr>
<td>take(5)</td>
<td>57.1% (12)</td>
<td>1</td>
<td>5</td>
<td>2.8</td>
<td>66.7% (8)</td>
</tr>
<tr>
<td>last</td>
<td>42.9% (6)</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>16.7% (2)</td>
</tr>
<tr>
<td>drop(10).drop(1)</td>
<td>41.7% (5)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
<tr>
<td>takeRight(6).takeRight(6)</td>
<td>38.1% (8)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>8.3% (1)</td>
</tr>
<tr>
<td>toMap</td>
<td>57.9% (11)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
<tr>
<td>map(_-1 -&gt; 1)</td>
<td>42.1% (8)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
<tr>
<td>zipWithIndex</td>
<td>26.3% (5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
<tr>
<td>map(_-1.toInt)</td>
<td>15.8% (3)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.0% (0)</td>
</tr>
</tbody>
</table>

Table 4.6: Frequently removed method sequences in the Syntax and GIM groups and their occurrence in the PBE group.
Chapter 5

A Formal Model for Interactive Synthesis

With the generalization of PBE offered with GIM in the previous section shown to be effective, our next goal is to investigate the theoretical foundation of interactive synthesis. To this end, we present a model of the iterative synthesis process, centered around the interaction between the synthesizer and a human user, and grounded in the theory of abstract domains [CC77]. This model aims to capture work with a wide array of user-driven synthesizers. We use this model to prove both existing properties of synthesizers and desirable properties in future synthesis tools. In order to do so, our definitions and results are grounded in real-world examples. This model provides us with a theoretical understanding of the properties of the interaction (e.g., progress, termination guarantees) which can then be applied to current and future synthesizers.

5.1 Introduction

Program synthesis is the problem of computing from a specification a program that implements it. The classic synthesis problem searches for an implementation to a full specification, usually encoded in some logic. Newer variations on the problem have turned to partial specifications, such as input-output examples or type information, that are easier for the user to provide but describe a much wider array of matching programs.

Synthesis tools for end-users are available for a wide variety of purposes from creating formulae in Microsoft Excel [Gul11] to formulating SQL queries [WCB17]. Programming by Example tools that accept input-output pairs as their specification have also matured enough to be practical on their own [Gul11, WCB17, LWDW01, LH95, WM93, Ant05, WLF15, OSY16, LG14] or as a way to refine the results of type-driven synthesis [OZ15, FCD15].

When the specifications are partial, the user is often brought into the loop to aid the synthesizer to determine the correctness of the final product and to direct it with additional feedback in case of ambiguity. Gulwani [Gul12] separates synthesizers by
their model of interaction with the user. Notably two categories are (i) *user-driven* synthesis tools, in which the user is responsible for verifying the artifact returned by the synthesizer, and if incorrect, for providing additional specifications to the synthesizer, and (ii) *synthesizer-driven* tools, in which the synthesizer poses the user with membership queries for ambiguous examples until it has reached a level of confidence high enough to return a program to the user as a validation query. Counterexample-guided Inductive Synthesis (CEGIS) [SLTB+06], in which a verifier is provided with a specification, and each program from the synthesizer is verified to produce either acceptance or an automatically generated counterexample, is seen as its own category, as the interactivity is between the synthesizer and the verifier.

**Interactive synthesis** Despite the fact that few user-driven tools define themselves as interactive synthesis tools, it is important to note that interactivity is always inherent in the synthesis workflow: the user provides some initial specification, runs the synthesis procedure, and is presented with an answer. However, they may not be satisfied with this answer, which leads to refinement of the specifications and another execution of the synthesizer. This iterative process of candidate solution and refinement is rarely discussed, as focus tends to remain on each single attempt to reach the user’s intended program with as partial a specification as possible, via rankings and biases.

**Interaction via predicates on programs** Likewise, while each synthesis tool usually treats the mode of specification it leverages as its own domain—input-output examples, types, etc.—the common ground is often overlooked. Each of these modes of feedback can be seen as a predicate over programs, and the process of providing a partial specification as constraining the space of possible programs to just those that satisfy each of the predicates. We remind the reader that an input-output pair \((i, o)\), often seen as the simple and natural tool for partial specification, can be seen as the predicate \([\text{program}](i) = o\). As Claim 4.3.1, as well as other previous work [DCSY17] have shown, examples are a weak tool with which to provide specification. The addition of other predicates in GIM (Chapter 4) allows for better separation between programs.

The most comprehensive specification that describes a target program \(m^*\) is every predicate available in the system that holds for \(m^*\). However, this complete specification is very likely infinite or not computable. On the other hand, an initial specification can be so partial as to rule out only a small number of programs in the candidate space. The solution is to leverage the user’s ability to compute what is incomputable. In the words of Knuth, “Some tasks are best done by machine, while others are best done by human insight; and a properly designed system will find the right balance” [Knu07] — by incrementally providing additional predicates to refine the specification, a process which at the limit will reach the comprehensive specification. In this chapter, we limit our scope to this form of iterative synthesis, where the set of predicates is built monotonically.
Properties of iterative synthesis  The attempts to get program synthesis tools to return a “suitable” program as soon as possible are based in heuristics and optimizations, and vary greatly from one tool to the next. Additionally, these tools often have no concept of convergence, or when the session will be forced to end, to explain their behavior should these heuristics fail. While a few of the tools have been modeled individually, in order to prove specific properties, the field is still lacking a generic model that can be used to prove properties such as termination and establish criteria for the designers of future synthesis tools to take into account as they design their new frameworks.

Existing work  Previous work has modeled single iterations of different flavors of synthesis [ABJ+15, PG15], and the counterexample-guided model of synthesis (CEGIS) [JS17, SL08]. The synthesizer-driven model of program synthesis [LPP+17] has also been modeled via predicates, where user answers to membership queries are translated into constraints and used to reduce the search space for the next iteration. A learner-teacher model of program synthesis [LMN16] has been presented mainly to model CEGIS, but can be applied to an iterative, user-driven synthesizer as well, with the human user taking on the role of the teacher. However, this model provides only guarantees stemming from the properties of the program space made available by the synthesizer, with little consideration of the way feedback is provided to the synthesizer. For a CEGIS model, this is sufficient, as communication between the teacher and learner is chiefly in examples, but is unsuitable for a more generic model where feedback formats and specification tools are multiform.

5.1.1 Our approach

In this chapter, we formulate a model for interactive synthesis using the theory of abstract domains.

An abstract domain of predicates  Given a domain of programs $M$ and a domain of predicates on programs $P$, we define the concrete domain of the synthesis algorithm to be sets of programs $(2^M, \subseteq)$ and the abstract domain to be sets of predicates $(2^P, \supseteq)$, with an abstraction function that produces the incomputable set of all predicates that hold for the set of programs, and a concretization function that produces the equally incomputable set of all programs that satisfy a conjunction of all predicates in a given set. Since both these sets are likely not computable, a real synthesizer relies on the synthesizer’s representation of the state to replace a concretization, and the user to replace the abstraction. Section 5.2 formally defines these domains and the operations on them.

Iterative, interactive synthesis  In this domain, we can then define an iterative synthesis algorithm as an iterative refinement (i.e., adding of predicates) of the speci-
ification in each iteration of the process. This creates a synthesizer state, in itself an abstract element, from which the next program displayed to the user as a candidate solution is selected. This process, in essence, is leveraging the user to compute the abstraction of the target program, or more accurately, a finite subset of it. If a finite subset that underapproximates the target exists, the synthesis session can converge regardless of the implementation. Section 5.3 defines an iterative synthesis session, the notion of progress by the user, and the terms for convergence.

**Properties of interactive synthesis** Using this model, we show several properties of interactive synthesis. In section 5.5 we define the point from which a synthesis session can no longer converge, even if the user has, from their point of view, only provided correct specifications, and properties of the point we must backtrack to when that happens. Section 5.4 offers two separate sets of limitations on the model that lead to convergence (i.e., a finite session) in every session. A *well-quasi-order of predicates* ensures that all sessions will terminate, and a *locally strongest user* condition ensures termination when predicates only have a well-founded-ordering. We demonstrate these conditions and properties using realistic examples.

**Implications** The implications of these properties for the designers of future synthesis tools will be discussed in Chapter 6.

### 5.1.2 Main contributions

The main contributions of this chapter are:

- A general model for iterative synthesis using the theory of abstract domains,
- Convergence conditions for iterative synthesis sessions, based on properties of the predicates and user behavior,
- Insights about backtracking when a session can no longer converge, and
- Recommendations for designers of future synthesis tools.

### 5.2 Foundations for Synthesis with Abstract Domains

In this chapter, we formalize interactive synthesis using abstract domains, where the role of the user is to strengthen the abstraction of the target program, while the role of the synthesizer is to concretize the abstraction and pick a concrete element from it as a candidate program. To do so, we start, in this section, by formalizing a single iteration that consists of a user providing a spec as input and the synthesizer returning a program.
Let us consider the domain of all programs, in all languages. Out of these, only a subset is available to the user via the synthesizer. We denote this, our program search space, \( M \subseteq U \).

User-driven synthesis is guided by the concept of a target program in the user’s mind. We denote \( U^* \subseteq U \) the set of programs that satisfy the user’s concept of a correct program, and \( M^* = U^* \cap M \), the subset of \( U^* \) that is in the synthesizer’s search space. A user’s intention is realizable if \( M^* \neq \emptyset \). It is important to notice for the remainder of this chapter that \( M^* \), while a subset of the synthesizer’s search space, is not actually known to the synthesizer.

In order to encode the specification, let us also consider a (possibly infinite) set \( P \) of predicates over programs. We assume that every \( p \in P \) is decidable. When considered against some set of programs \( T \), each predicate \( p \in P \) defines a subset of programs from \( T \) that satisfy it, denoted \( \{ m \in T \mid m \vDash p \} \). In this way, the same set of predicates \( P \) can define subsets of both \( M \) and \( U \). In this sense, the predicates can be viewed as formulas, and the programs as structures. We do not, however, assume or use any internal structure of the predicates in this chapter.

In particular, we will use predicates in implication modulo a theory of programs. We write \( p \Rightarrow_T q \) to denote \( \forall m \in T. \ m \vDash p \Rightarrow m \vDash q \). The same extends to a set of predicates, \( A \Rightarrow_T q \), to mean as their conjunction.

The remainder of this chapter assumes working with a specific \( P \) and a specific \( M \), and that the user is seeking a specific \( M^* \). This means all the definitions that follow are parametric in \( M \) and \( P \), and when used also in \( M^* \).

5.2.1 An abstract domain for programs

Our concrete domain consists of the powerset lattice \( (2^M, \subseteq) \) (where the least element is \( \emptyset \) and the greatest element is \( M \)). That is, each concrete element is a set of programs, and the sets get smaller when lower in the lattice.

During the synthesis process, the synthesizer represents (or abstracts) sets of programs from the concrete domain using sets of predicates from \( P \). Formally, let \( A = 2^P \). The synthesis process uses an abstract domain that consists of the powerset lattice \( (A, \subseteq) \), where \( \subseteq \) is defined as \( \supseteq \). That is, each abstract element is a set of predicates (interpreted as a conjunction), and the sets get larger (or more constrained) when lower in the lattice. Join, meet, bottom, and top are defined as they usually are in the powerset domain: For two abstract elements \( A_1, A_2 \in A \), meet is defined as \( A_1 \cap A_2 = A_1 \cup A_2 \) and join as \( A_1 \cup A_2 = A_1 \cap A_2 \). Further, \( \top = \emptyset \) and \( \bot = P \).

From here on, we refer to \( A \in P \) as elements in the lattice and as sets of predicates interchangeably. Which one we mean should be clear from the context (e.g., the operators used).

69
Galois connection  We would like an abstract element $A \in \mathcal{A}$ to represent the set of programs $s \in M$ for which every predicate $p \in A$ holds. To do so, we define a Galois connection between $(\mathcal{P}, \subseteq)$ and $(\mathcal{A}, \sqsubseteq)$.

**Definition 5.2.1 (Abstraction).** For a single program $m \in M$, we define the abstraction function $\beta(m) = \{ p \in \mathcal{P} \mid m \models p \}$, which abstracts $m$ into the set of all predicates that hold for $m$. From this we define for a set of programs $C \subseteq M$ the abstraction $\alpha(C) = \bigsqcup_{m \in C} \beta(m) = \{ p \in \mathcal{P} \mid \forall m \in C. m \models p \}$.

This is similar to the abstraction performed by Houdini [FL01], Daikon [EPG+07], and $D^3$ [PSY16].

**Definition 5.2.2 (Concretization).** For an abstract element $A \in \mathcal{A}$, we define the concretization function $\gamma(A) = \{ m \in M \mid \forall p \in A. m \models p \}$, or all programs for which all constraints in $A$ hold.

It is easy to verify that $(\alpha, \gamma)$ is a Galois connection.

Recall that in the abstract domain, $\perp = \mathcal{P}$ and $\top = \emptyset$. Therefore, $\gamma(\top) = M$, which means that the top element represents all valid programs in $M$, as desired. On the other hand, $\gamma(\perp)$ is not necessarily the empty set, since there might be valid programs that satisfy all predicates in $\mathcal{P}$. However, in the typical case, $\mathcal{P}$ contains contradicting predicates (e.g., a predicate and its negation, or examples mapping the same input $i$ to different outputs $o_1 \neq o_2$), in which case $\gamma(\perp)$ represents an empty set of programs.

Reducing the search space  The non-interactive, single-step, synthesis problem can now be described as one for which the input is a (partial) specification of the target program in the form of an abstract element $A \in \mathcal{A}$, and the output is some program from the set of programs it describes. The selection is (usually) not random, but rather influenced by internal representation in the synthesizer, as well as ranking functions. To reason about the synthesizer’s role, we define $\text{Select} : \mathcal{A} \to M \cup \{ \perp \}$, the synthesizer’s operation of finding such a program. $\text{Select}(A)$ amounts to picking a concrete element from $\gamma(A)$, or returning $\perp$ if no such element exists; hence, it can be understood as partially concretizing the abstract element. The implementation of $\text{Select}$ is dependent on the synthesis algorithm being used.

**5.2.2 Examples**

**Type-directed synthesis as an abstract domain**  A widely used domain of predicates is a domain of type information. When creating a procedure via type-directed synthesis, the specification to the synthesis procedure is provided via type predicates for the procedure’s formals $(\text{name}, \tau) \in \text{Formals} \times \mathcal{T}$ and a desired return-type predicate, $\tau_{\text{ret}} \in \mathcal{T}$ which will hold according to the $\subseteq$ relation on types. A similar specification is used for type-directed synthesis that produces code snippets: the same $\tau_{\text{ret}}$ specifies
the target type (usually assigned to a variable) and the available variables are specified using type predicates \((\text{name}, \tau) \in \text{Vars} \times \mathcal{T}\) for local variables \(\text{Vars}\).

**PBE as an abstract domain**  
Another frequently used domain of predicates is the domain of input-output examples. Recall that each program \(m\) defines a function, \([m] : I \rightarrow O \cup \{\perp\}\), that maps inputs to outputs (or to error). Programming by example considers the predicates \(\mathcal{P} = I \times O\), where each pair \((\iota, \omega) \in \mathcal{P}\) dictates that for input \(\iota\), the program outputs \(\omega\). For this purpose, we define 
\[m \models (\iota, \omega) \iff [m](\iota) = \omega.\]

**Syntactic feedback as an abstract domain**  
Chapter 4 introduces a domain of predicates that provide syntactic restrictions on programs, intended for use by programmers: The \(\text{retain}(f \cdots g)\) predicate which holds only for programs that make use of a function or operator sequence \(f \cdots g\), or \(\text{remove}(f \cdots g)\) which holds only when they do not. For linear functional programs, these operators can also be generalized to general subsequences—\(\text{exclude}(f \cdot g)\) will hold for programs where there are no \(i < j\) s.t. \(f_i = f, f_j = g\). Additionally, these can be generalized to partial or full subtrees. The predicates used in this chapter are limited to these, but in several examples in this chapter we make use of predicates suggested by or simply in the spirit of those shown in Chapter 4.

**5.2.3 Computability of the model**

We notice that, in general, both \(\alpha\) and \(\gamma\) are non computable: \(\alpha\) because \(\mathcal{P}\) may be infinite; and even though any \(A\) provided by the user will always be a finite set, \(\gamma\) may still not be computable as a finite set of predicates may return an infinite subset of an infinite \(M\). Because of that, neither of them is used directly by any concrete implementation of the model. Concretization is only performed as part of a \(\text{Select}(A)\) operation, representing the synthesizer’s generation of a program based on its description of the reduced program space \(A\), which need not actually create the concrete set of programs represented by \(A\). In synthesizers based on version space algebra (VSA) [LWDW03], for instance, only a representation of the space of all programs is constructed, from which a single concrete program is then selected.

Abstraction is also never performed by the algorithm, but rather by the user: the target programs, \(M^*\), as envisioned by the user, are described in the input specification \(A\) by the selected predicates. This is less precise than a full (and possibly infinite) \(\alpha(\{m^*\})\) of some \(m^* \in M^*\), but in an iterative synthesis process can be refined by the user when the result is insufficient, which means that the synthesizer state (representing the accumulated user input) comes closer to \(\alpha(\{m^*\})\) with each iteration. (Note that unlike a classical abstraction framework [CC77], where it is important to soundly abstract the entire set \(M^*\), in synthesis it suffices to abstract some nonempty subset of \(M^*\).)
**Intuition**  If the user could produce a full specification $S^* \subseteq P$ (or as full as $P$ allows), satisfying it could be a matter of arbitrarily selecting any program from $\gamma(S^*)$. However, since creating full specifications is hard or even impossible, the process of interactive synthesis, which will be described in the next section, is essentially building up to a fuller specification in every iteration. The user adds new specifications to rule out each undesirable candidate program, and the meet operation collects added specifications into the synthesizer state, which at the limit will reach $S^*$.

### 5.3 An Abstract Model of Interactive Synthesis

Section 5.2 discussed a model for a single iteration of synthesis. We now wish to describe the iterative process that exists, even if implicitly, in most synthesizers. In it the user will keep adding to the specifications given every time the synthesis procedure offers an unsatisfactory candidate. We formulate this as questions (candidate programs) and answers (additional specifications).

**Definition 5.3.1 (Synthesis session).** A synthesis session is a sequence of steps by the user and synthesizer $S = (A_0, q_1), (A_1, q_2), \ldots$ such that $q_i \in M \cup \{\bot\}$ are synthesizer questions and $A_i \in \mathcal{P}_{\text{fin}}(P) \cup \{\bot\}$ are user answers, where $\mathcal{P}_{\text{fin}}(P)$ is the set of all finite subsets of $P$ and $\bot$ signifies a forced contradiction. We denote $A_0$ the initial specifications provided by the user.

Within a synthesis session we define the state of the synthesizer via the constraints on it provided by the user, as follows:

**Definition 5.3.2 (Synthesizer state).** The state of the synthesizer $S \subseteq P$ is an abstract element describing the portion of the search space requested by the user. If the user has given feedback for $i$ iterations in the form of the elements $A_0, A_1, \ldots, A_i \subseteq P$, the state after $i$ iterations of feedback is $S_i = \bigcup_{0 \leq j \leq i} A_j$.

Interactive synthesis can now be formalized as a process in which both the state of the synthesizer and the interaction between the synthesizer and the user are based on abstract elements. In step $i$, the synthesizer selects a program $q_i \in M$ using $\text{Select}(S_{i-1})$, and poses $q_i$ as a validation question to the user. The user accepts or rejects the program. In case of rejection, the user responds with an answer $A_i \in A$ in the form of an abstract element which consists of one or more predicates out of $P$. Given the user’s answer $A_i$, the new state of the synthesizer in step $i + 1$ is set to $S_{i+1} = S_i \cap A_{i+1}$, thus narrowing the set of concretizations to consider. Or, in other words, we can now define $S_{i+1} = \bigcap_{0 \leq j \leq i+1} A_j$. The search is over either when $\text{Select}$ returns nothing because $\gamma(S_i) = \emptyset$ and represents no programs, or when the user is satisfied and accepts the program.

Notice that, unlike the classical use of abstraction, where the intent is to describe as many concrete states as possible, and so new information is appended via join, here our purpose is to refine, and so we use meet.
Synthesis users  In order to reason about iterative synthesis, we must define the user’s behavior. We have already defined $U^*$ the set of programs in $U$ the user is willing to accept, as well as $M^*$, the intersection between the user’s concept of the target program and the search space of the synthesizer. We now add guarantees for the iterative behavior:

**Definition 5.3.3** (User correctness). A user step, providing $A_i$ as an additional specification, is *correct* when $A_i \subseteq \{ p \in \mathcal{P} | \exists m \in U^*.m \models p \}$.

Correctness means the user will not provide predicates that are inconsistent with their idea of the target. Notice that this set of predicates may still contain a contradiction, as it contains predicates of different programs, and that even if no explicit contradiction exists, subsets of it may still evaluate to $\emptyset$ over the domain $M$.

According to definition 5.3.3, a correct user may still provide predicates that hold for some, but not all, of $U^*$. This may seem unintuitive, but realistically occurs because (a) $U^*$ may not be sufficiently described with predicates from $\mathcal{P}$, but a subset of it may, (b) given a current candidate program $m$, the user sets a trajectory for the synthesis procedure and makes local decisions that may rule out some programs in $U^*$, or conflict with other (similarly local) decisions made in the past.

**Definition 5.3.4** (Synthesis user). The *behavior of the user* includes the following guarantees:

1. The user is correct for as long as they can be. If the user can no longer provide an answer that is correct, they will answer $\perp$.

2. If a user sees a program in $M^*$, they will accept it.

Finally we define a feasible synthesis session as a session that can be reached by the actions of a user and a synthesizer:

**Definition 5.3.5** (Feasible synthesis session). A *feasible synthesis session* is a synthesis session $\mathcal{S} = (A_0, q_1), (A_1, q_2), \ldots$ that satisfies the following:

(a) All $A_i$ are correct steps (definition 5.3.3) or $\perp$,

(b) $q_i = \text{Select}(S_{i-1})$, i.e. $q_i \in \gamma(S_{i-1}) \cup \{\perp\}$, where $\perp$ signifies no possible program,

(c) If $q_n \in M^* \cup \{\perp\}$ then $\mathcal{S}$ is finite and of length $n$, and

(d) In a finite $\mathcal{S}$ of length $n$, $q_n \in M^* \cup \{\perp\}$

where item (b) is a requirements for synthesizer correctness, and items (a), (c) and (d) are requirements for user correctness.
Remember that additionally, from the definition of \textit{Select}, if $S$ is finite of length $n$, then $q_n = \bot \iff \gamma(S_{n-1}) = \emptyset$.

These mean that a feasible synthesis session is either (i) infinite, (ii) ends by returning $\bot$, (iii) or ends with the user accepting $q_n$ the last program.

For the remainder of this chapter we are only interested in feasible synthesis sessions.

**Definition 5.3.6** (Convergence). A synthesis session $(A_0, q_1), (A_1, q_2), \ldots, (A_{n-1}, q_n)$ is said to converge if $\gamma(S_n) \subseteq M^*$. It has converged successfully if $\gamma(S_n) \neq \emptyset$.

When a session has terminated with any result other than $\bot$, this will mean that the user accepts $q_n$, but convergence is in fact a stronger condition. This is because definition 5.3.5((d)) can refer to a case where the synthesizer has offered a program out of $M^*$ at any point in the session, because of the implementation of \textit{Select}, ranking, or domain knowledge, thereby causing the session to end immediately. Convergence, on the other hand, ensures that regardless of the implementation of \textit{Select}, a program from $M^*$ will be returned (or no program at all). This definition reflects the fact that, unlike classical abstraction frameworks, where one seeks an overapproximation of the target that is “precise” enough, convergence of a synthesis procedure requires an \textit{underapproximation} of the target. For convergence to be successful, that underapproximation must be nonempty. For the remainder of this chapter we will be mostly interested in the worst-case implementation of \textit{Select}, where the session either converges or is infinite.

### 5.3.1 Progress-making sessions

The first basic property needed in order to explore convergence is that the synthesis session is progressing—refining not only the abstract element of the synthesizer state but also its concretization in the program space. We consider two kinds of progress, weak and strong, which differ by the effect of the step on the synthesizer state. Section 5.4 will leverage progress into results on termination.

**Definition 5.3.7** (Weak progress). A user answer $A_i$ is said to create weak progress in iteration $i$ of a synthesis session if $\gamma(S_{i-1} \cap A_i) \subset \gamma(S_{i-1})$. This means that $A_i$ has ruled out at least one program from $M$ described by $S_{i-1}$.

We say a synthesis session makes weak progress if every user answer $A_i$ in the session makes weak progress.

Note that it is not enough to demand that $S_{i-1} \cap A_i \sqsupset S_{i-1}$: the user can provide a predicate $p$ that rules out no program in $\gamma(S_{i-1})$, which means $\gamma(S_{i-1}) = \gamma(S_i)$ but since it was not given before by the user, $S_{i-1} \cap A_i \sqsupset S_{i-1}$.

**Lemma 5.3.8** (Weak progress by implication). User answer $A_i$ in iteration $i$ of synthesis session makes weak progress if and only if $S_{i-1} \not\supseteq_M A_i$. 

74
Proof. Let \( S \) be a synthesis session. Step \( i \) makes weak progress \( \iff \gamma(S_{i-1} \cap A_i) \subseteq \gamma(S_{i-1}) \iff \exists m \in M. m \in \gamma(S_{i-1}) \setminus \gamma(S_{i-1} \cap A_i) = \gamma(S_{i-1}) \cap \gamma(A_i) \iff \exists m \in M. m \in \gamma(S_{i-1}) \setminus \gamma(A_i) \iff \exists m \in M. \forall p \in S_{i-1}, m \models p, \exists p \in A_i. m \not\models p \iff \exists p \in A_i. S_{i-1} \neq_{M} p \iff S_{i-1} \neq_{M} A_i.

Lemma 5.3.8 gives us a test for the synthesizer to apply should the creators of the synthesizer wish for it to enforce progress in every iteration.

Example 5.3.9. Let us examine predicates used for providing positive feedback. In PBE this might be an example that reinforces some behavior that is good in the current program. In other predicates this might be okaying a syntactic portion on the program, or in other words, asking the synthesizer to keep something for future programs. Another option is approving of an intermediate value of the program for a specific input—something which holds for the current program.

All of these, while not ruling out the current program, may rule out other programs in the space. This means that in a synthesizer which enumerates the entire space of \( M \) in some order, the same \( q_i \) will be displayed as \( q_{i+1} \). However, since the portion of the program space represented by \( S_n \) is different, some implementations of Select may return a different program.

Definition 5.3.10 (Strong progress). A user answer \( A_i \) is said to create strong progress in the synthesis session if \( q_i \not\in \gamma(S_{i-1} \cap A_i) \), or in other words, if \( \alpha(\{q_i\}) \not\subseteq A_i \).

We say a synthesis session makes strong progress if every user answer in the session makes strong progress.

Definition 5.3.10 is stronger than that defined in definition 5.3.7 as it ensures the user will not be shown the same program again, regardless of the implementation of Select. If Select has some preference bias—such as an ordering over the programs—then non-strong progress will essentially lead to the same program being returned; however, we do not preclude the general case where changing the specification in any way or even just re-running the synthesizer may yield a different program.

Example 5.3.11. The FlashFill implementation in Microsoft Excel [Gul11] allows only predicates that would cause strong progress. Specifically, as the program candidate in each iteration of FlashFill is executed on the entire dataset and the results are shown to the user. The user can then make changes to records where the result of the executed program is not the desired result. This mean that the set of predicates available to the user at iteration \( i \) is not any \( \{(r, o) \mid r \text{ is a record in the table}\} \), but only \( \{(r, o) \mid \|q_i\|(r) \neq o\} \). Since every \( p \in A_i \) necessarily rules out \( q_i \), this is an even stronger requirement than that of strong progress in definition 5.3.10.

Due to our assumption on the user correctness, the strong progress requirement can be equivalently formulated by requiring the user to use at least one predicate that differentiates \( q_i \) from \( M^* \):
Definition 5.3.12 (Diff). We define the diff between two programs $m_1, m_2 \in M$ in the program space over the set of available predicates to be $\text{diff}(m_1, m_2) = \{p \in \mathcal{P} \mid m_2 \models p \land m_1 \not\models p\} = \beta(m_2) \setminus \beta(m_1)$.

Lemma 5.3.13 (Correct strong progress by differentiating predicate). A correct user answer $A_i$ in iteration $i + 1$ of a synthesis session makes strong progress if and only if $A_i \cap \bigcup_{m \in M^*} \text{diff}(q_i, m) \neq \emptyset$.

While progress is a natural requirement to make, it may not always be obtainable with the available predicates. There may simply not be predicates with which to rule out the current program, for instance, but, most often, there is simply no correct step with which to continue the session. Next, we define the result of the clash between progress and correctness and demonstrate a scenario where it manifests:

Definition 5.3.14 (Non-progress point). Iteration $i$ is a weak non-progress point (resp. strong non-progress point) if any predicate $p$ that would cause weak (resp., strong) progress is incorrect, i.e., $\forall m \in U^*. m \not\models p$.

In the sequel, we simply refer to a “non-progress point” since the weak/strong qualifier is determined by the kind of interaction enforced by the synthesizer.

If iteration $i$ is a non-progress point, then by correctness the user is forced to answer $\perp$. In practice, this means iteration $i + 1$ will necessarily be $(\perp_P, \perp_M)$.

Example 5.3.15. Consider a domain of programs and a set of predicates $\mathcal{P} = \{\text{exclude}(f) \mid f \in \mathcal{V}\} \cup \{\text{include}(f) \mid f \in \mathcal{V}\}$ over some vocabulary of methods $\mathcal{V}$. The user is looking for a program that will provide them with the second element of a list of strings. Let us assume that $U^* = M^* = \{\text{input.tail.head}\}$, and that the user is shown $q_i = \text{input.head.tail}$.

If the current synthesizer enforces strong progress, the user is now at an impasse: includes are a form of positive feedback, approving of something in the current program. While they may rule out some program in the synthesizer state, they will not rule out $q_i$. However, with the given set of predicates, either option that will make any progress, $\text{exclude(head)}$ and $\text{exclude(tail)}$, will violate correctness, and will cause $S_i \cap M^* = \emptyset$.

5.4 Termination

In general, a synthesis session may never terminate. For instance, it is easy to show using this model that PBE may never terminate: let us assume the user is searching for a program where conversion from polar to cartesian coordinates is required. The user will provide some examples for desired input-output pairs, and a program that applies the sine function to implement the conversion will be part of the synthesizer state, but
no matter how many examples are provided, there will still be programs that use some interpolated polynomial instead of sine, thereby keeping $\gamma(S_i)$ from ever reaching $M^*$.

We now show two conditions for termination for synthesizers, based on properties of their predicates. The first is a condition for both strong and weak progress sessions, demanding a strong requirement from the synthesizer, a well-quasi-ordering of the predicates. The second is a condition for synthesis sessions that make strong progress, and is modeled on a property similar to well-quasi-order’s finite basis property. In it, we can weaken the requirement on the predicates, but in exchange add a requirement from the user.

### 5.4.1 WQO predicates

We first show that termination can be guaranteed using the theory of well-quasi-ordering:

**Definition 5.4.1** (Well-quasi-order [Kru60]). Let $\preceq$ be quasi-order on $X$ (i.e., $\preceq \subseteq X \times X$ is a reflexive and transitive relation). By convention, $x > y$ denotes $y \preceq x$ $\land$ $x \not\preceq y$.

The following definitions are equivalent:

1. $\preceq$ is a wqo over space $X$.
2. In every infinite sequence $x_1, x_2, \ldots$ there exist $i < j$ s.t. $x_i \preceq x_j$, and
3. $X$ satisfies both: (a) every sequence $x_1 > x_2 > \ldots$ is finite (the strictly descending chain condition, also known as well-foundedness), and (b) every sequence $x_1, x_2, \ldots$ with $x_i \not\preceq x_j$ for $i \neq j$ is finite (the incomparable chain condition, also known as the antichain condition).

**Theorem 5.1.** Let $p \triangleleft p' \iff p \Rightarrow_M p'$. If $\preceq$ is a well-quasi-ordering over the set $\cup_{m' \in M^*, \beta(m')} \beta(m')$, then any synthesis session that makes (weak or strong) progress will always converge in a finite number of steps.

**Proof.** Since every strong progress session also makes weak progress, it suffices to prove the theorem for weak progress sessions.

Let us assume, by way of contradiction, that $S$ is an infinite synthesis session that makes weak progress. We construct the infinite sequence $p_0, p_1, \ldots$ such that $p_i$ is some progress-making predicate from $A_i$. Since $S$ makes weak progress, we know that $S_{i-1} \not\Rightarrow_M p_i$ (Lemma 5.3.8) and in particular, for every $p' \in S_{i-1}$, $p' \not\Rightarrow_M p_i$. From definition 5.3.2, $\forall j, j < i \Rightarrow (p_i \not\Rightarrow_M p_j)$, or in other words, $\forall j, j < i \Rightarrow (p_i \not\preceq p_j)$. But since $\preceq$ is a wqo, in every infinite sequence $\exists i, j, i < j$ $\land$ $p_i \preceq p_j$ (from definition 5.4.1(b)), leading to a contradiction. This means a session must be finite, i.e. converge.

From this, if the entire predicates set $P$ is a wqo, then the synthesizer will terminate for every $M^*$. 

77
Example 5.4.2. While it is easy to see that examples are not a wqo, as the entire domain is incomparable, there are domains of predicates that do create a wqo. For instance, a family of syntactic predicates \( \text{exclude}(f_1 \cdots f_2 \cdots f_n) \) that exclude programs containing a specific subsequence of function calls (not necessarily consecutive) will be a wqo over the domain of linear programs \([\text{Hig52}]\). In this domain, a user can express feedback such as \( \text{exclude}(\text{close} \cdots \text{read}) \), thereby ruling out every program that creates a read-after-close error.

5.4.2 Locally strongest user

In this subsection, we relax the well-quasi-order requirement on the predicates, and prove another termination property by assuming some locally-optimal property of the user.

Definition 5.4.3 (Base set). Let \( S \subseteq \mathcal{P} \) be a set of predicates. We define the base of \( S \), \( \text{Base}(S) = \{ p \in S \mid \forall p', p' \Rightarrow_M p \Rightarrow p = p' \} \), i.e. the set of strongest predicates in \( S \).

In order to simplify we assume \( \mathcal{P} \) does not contain equivalent predicates.

Let us now add a new restriction on the user, which strengthens the strong progress requirement of the synthesizer:

Definition 5.4.4 (Locally strongest user). Given a candidate program \( q_i \notin M^* \), a locally strongest user will answer with \( A_i \) such that \( A_i \cap \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \neq \emptyset \). That is, at least one predicate in the answer \( A_i \) will be taken from \( \text{Base}(\text{diff}(q_i, m')) \) of some target program \( m' \) (where the latter means that no stronger predicate exists in \( \text{diff}(q_i, m') \)).

In other words, a locally strongest user will always make progress using the most effective (i.e., strongest) predicates available. This means that, for instance when using GIM predicates (Section 4.4.1), given a choice between two sequence exclusion predicates \( \text{exclude}(\text{drop}) \) and \( \text{exclude}(\text{drop} \cdot \text{take}) \), if they are both relevant, the user will select the one making more impact – which is the sensible choice, as excluding the subsequence when the individual function is undesirable could cause it to appear again.

We notice that in case the sets of predicates in question have an infinitely decreasing (i.e., infinitely getting stronger) sequence of predicates, this restriction on the user is at odds with correctness: no predicate from the infinite decreasing sequence will be represented in its base set, which means the user may have a correct predicate available to them from \( \bigcup_{m' \in M^*} \text{diff}(q_i, m') \) but no action in the union on the base sets.

To counteract this, we would like to make sure every chain of predicates would have a strongest element to add to the base set. We therefore add a requirement for \( \bigcup_{m' \in M^*} \beta(m') \) to be a well-founded order: we recall that if \( X \) is a wfo, it satisfies the strictly descending chain condition in definition 5.4.1(c) (but not necessarily the incomparable chain condition). The following lemma shows that if \( \bigcup_{m' \in M^*} \beta(m') \) is a...
wfo, then a correct user that is able to make strong progress can also be locally strongest, i.e., it will never get stuck due to inability to find a “strongest” predicate.

**Lemma 5.4.5.** Let \( p \preceq p' \iff p \Rightarrow_M p' \). If \( \preceq \) is a wfo over \( \bigcup_{m' \in M^*} \beta(m') \), then whenever \( \bigcup_{m' \in M^*} \text{diff}(q_i, m') \neq \emptyset \), we have that \( \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \neq \emptyset \) as well.

**Proof.** First note that if \( \preceq \) is a wfo over \( \bigcup_{m' \in M^*} \beta(m') \), then it is also a wfo over \( \bigcup_{m' \in M^*} \text{diff}(q_i, m') \) for any \( q_i \notin M^* \). This is immediate from the property that \( \text{diff}(q_i, m') \subseteq \beta(m') \) and hence \( \bigcup_{m' \in M^*} \text{diff}(q_i, m') \subseteq \bigcup_{m' \in M^*} \beta(m') \). Since \( \bigcup_{m' \in M^*} \text{diff}(q_i, m') \) is nonempty, well foundedness ensures that its base set is also nonempty, and hence also \( \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \neq \emptyset \).

We can now formalize our termination result for a locally strongest user. We start with the simpler case where \( M^* \) is a singleton set, and then extend it to the general case.

**Theorem 5.2.** If \( \bigcup_{m' \in M^*} \beta(m') \) is a wfo, \( \bigcup_{m' \in M^*} \text{Base}(\beta(m')) \) is finite and the user is locally strongest, then any synthesis session that makes strong progress will converge in a finite number of steps.

Notice that when using \( \Rightarrow_M \) as an order relation, the requirement of finiteness of \( \bigcup_{m' \in M^*} \text{Base}(\beta(m')) \) is similar to a wqo’s finite basis requirement ([Higman [Hig52]]). However, this requirement is only applied to \( \beta(m') \) for \( m' \in M^* \), not to all sets, and does not require an upwards-closed set. Also notice that if \( \bigcup_{m' \in M^*} \beta(m') \) was a wqo, as required from theorem 5.1, this would already be true because of the finite basis property.

**Proof.** First we show that \( \text{Base}(\text{diff}(q_i, m^*)) \subseteq \text{Base}(\beta(m^*)) \) for every \( m^* \in M^* \) and \( q_i \in M \). Let us assume, by way of contradiction, that there exists a predicate \( p \in \text{Base}(\text{diff}(q_i, m^*)) \), \( p \notin \beta(m^*) \). We know that \( p \in \beta(m^*) \), since \( \text{diff}(q_i, m^*) \subseteq \beta(m^*) \), so for \( p \) to not be in \( \text{Base}(\beta(m^*)) \) there must be \( p' \in \text{Base}(\beta(m^*)) \) s.t. \( p' \Rightarrow_M p \). \( p' \) is not in \( \text{diff}(q_i, m^*) \), or it would also be in \( \text{Base}(\text{diff}(q_i, m^*)) \) instead of \( p \), which means that \( q_i \not\models p' \). However, since \( q_i \not\models p \) and \( p' \Rightarrow_M p \), we have reached a contradiction. This trivially implies that \( \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \subseteq \bigcup_{m' \in M^*} \text{Base}(\beta(m')) \), and hence finiteness of \( \bigcup_{m' \in M^*} \text{Base}(\beta(m')) \) ensures that \( \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \) is finite as well.

Next we see that since \( \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \) is finite, then if the user makes strong progress by selecting a predicate from \( \bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m')) \) in each iteration, the session will always converge in at most \( n \leq |\bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m'))| \) iterations when one of the following will occur:

- \( \gamma(S_n) \subseteq M^* \) (as will be seen later in definition 5.5.1, \( S_n = B \in B \)), and the session has converged successfully, or
- \( \gamma(S_n) = \emptyset \), which means \( q_{n+1} = \bot \), or the session has converged unsuccessfully.

79
The first option is a successful convergence. The second option, in which the session fails to converge successfully, is possible for two reasons. First, because our requirement for the user is not to select only from $\bigcup_{m' \in M^*} \text{Base}(\text{diff}(q_i, m'))$, and other correct user actions may still lead to a contradiction. Second, throughout the session, the user may select predicates from $\text{Base}(\text{diff}(q_i, m')) \subseteq \beta(m')$ of a different $m'$, and these predicates may contradict. The latter is no longer a possibility if $M^*$ is a singleton set.

**Example 5.4.6.** Let us assume a singleton $M^* = \{m^*\}$, a domain of functional programs over a vocabulary $V$ and a set of syntactic predicates $P = \{\text{include}(\text{seq}), \text{exclude}(\text{seq})\}$ predicates over all continuous sequences of methods $seq = f_1 \cdot f_2 \cdots f_n \in V$.

We can see immediately that $P$ itself is not a wfo: for every sequence used by $\text{include}$, there is a stronger predicate which includes a subsuming sequence. However, a specific target program $m^*$, and its description $\beta(m^*)$, is a different matter. While $\text{exclude}$ sequences can longer than the length of $m^*$ as long as we wish and will still appear in $\beta(m^*)$, $\text{include}$ sequences that are longer than $m^*$ will rule out $m^*$. This means that the chain of $\text{include}$ predicates in $\beta(m^*)$ is finite, and so $\beta(m^*)$ has a well-founded ordering.

### 5.5 Successful Convergence and Backtracking

In this section we characterize the cases where a synthesis session may converge successfully, in the sense that the user has a path that leads to successful convergence. We then examine situations in which a synthesis session trying to achieve a realizable target program goes awry and fails to converge successfully. The expected user behavior in these cases is to backtrack — to remove some of the provided specification or to cancel recent steps. We show that the point of realization that backtracking is needed is in many cases farther along the session than the point which necessitates backtracking. We explore the amount of sufficient backtracking, and show that it may be of any length.

Recall that a user’s intention is realizable if $M^* \neq \emptyset$ (see Section 5.2). We observe that this is a necessary condition but in general not sufficient, and successful convergence requires a stronger notion of realizability. To formalize this notion, we need the following definition:

**Definition 5.5.1 (Core set).** We say that a set $B \subseteq P$ is a complete specification if $\emptyset \neq \gamma(B) \subseteq M^*$. We define the core set of the synthesis problem as the set of all finite specifications, $B = \{B \subseteq P \mid \emptyset \neq \gamma(B) \subseteq M^* \land |B| \in \mathbb{N}\}$.

If there exists no $B \in B$ such that $\emptyset \neq \gamma(B) \subseteq M^*$, then there is no finite underapproximation of the target space in the abstract domain defined by $P$. In this situation, every synthesis will always fail, even if the specification is technically realizable. Based on this observation, we define a stronger notion of realizability:

**Definition 5.5.2 ($P$-realizability).** We say that $M^*$ is $P$-realizable if $B \neq \emptyset$. 

80
Indeed, \( \mathcal{P} \)-realizability is a necessary condition for successful convergence. For example example 5.3.15 describes the case in which the available predicates are syntactic predicates on a single function. If all programs in \( M \) that implement the user’s intention are of length 2 or more, then there may not be an underapproximation of \( M^* \). Likewise, when working with examples it may take infinitely many examples to differentiate between two programs (as shown in section 5.4), which means that the space described by any finite number of examples will still contain some program outside of \( M^* \).

Even with \( \mathcal{P} \)-realizability, the user’s steps may lead to a point where successful convergence is no longer possible. Next we generalize the above condition to refer to any point along the session. Furthermore, we show that the general condition is not only necessary but also sufficient for successful convergence (i.e., the user has a possible path to it). In order to provide the general condition we first define a property of the synthesizer’s state that captures situations where successful convergence is out of reach.

**Definition 5.5.3 (Inevitable failure point).** Let \( \mathcal{S} \) be a session. The state \( S_i \) is called an **inevitable failure point** if \( \forall B \in \mathcal{B}. \, \gamma(S_i) \cap \gamma(B) = \emptyset \).

In particular, if \( \gamma(S_i) \cap M^* = \emptyset \), then \( S_i \) is a point of inevitable failure. However, in general, this may not be the case — valid programs may exist even at an inevitable failure point (such programs are not contained in any \( B \in \mathcal{B} \)).

We note that the condition of an inevitable failure point can be equivalently defined as \( \forall B \in \mathcal{B}. \, \gamma(S_i) \not\supseteq \gamma(B) \). Clearly, an empty intersection of \( \gamma(S_i) \) with (the nonempty) \( \gamma(B) \) implies that \( \gamma(S_i) \) is not a superset of \( \gamma(B) \). For the other direction, if there exists \( B \) such that \( \gamma(S_i) \cap \gamma(B) \neq \emptyset \), then by taking the finite set \( B' = S_i \cup B \) we get \( \gamma(B') = \gamma(S_i) \cap \gamma(B) \subseteq \gamma(S_i) \). Moreover, \( \gamma(B') \) is nonempty and included in \( \gamma(B) \subseteq M^* \), hence \( B' \in \mathcal{B} \).

**Theorem 5.3 (Successful convergence).** Let \( \mathcal{S} \) be the prefix of length \( n \) of a synthesis session. Then the following conditions are equivalent:

1. \( S_{n-1} \) is not an inevitable failure point,
2. there exists a session \( \mathcal{S}' \) that extends \( \mathcal{S} \) and converges successfully.

**Proof.** The proof uses the equivalent formulation of inevitable failure point.

2 \( \Rightarrow \) 1 If \( \mathcal{S}' \) converges successfully at step \( m \), we select its final state \( S_{m-1} \) to be \( B \). Because of the successful convergence, \( \emptyset \neq \gamma(S_{m-1}) \subseteq M^* \), and since \( S_{m-1} \subseteq S_{n-1} \), then \( \gamma(S_{m-1}) \subseteq \gamma(S_{n-1}) \) (Galois connection).

1 \( \Rightarrow \) 2 Since \( S_{n-1} \) is not an inevitable failure point, there exists some \( B \) such that \( \gamma(S_{n-1}) \supseteq \gamma(B) \). Since \( B \) is finite, the user can answer with \( A_n = B \). Adding the step \( A_n \) leads to successful convergence: \( S'_n = S_{n-1} \cap A_n = S_{n-1} \cap B \), so \( \gamma(S'_n) = \gamma(S_{n-1}) \cap \gamma(B) = \gamma(B) \).
We note that unless \( q_n \in M^* \) (in which case the prefix \( S \) is complete), the extension \( S' \) of \( S \) constructed from the non-inevitable failure point by selecting \( A_n = B \) constitutes both weak and strong progress. The reason is that for \( q_n \notin M^* \), \( q_n \not\vDash B \), which makes this step a strong progress step, and, since some program has been eliminated, also a weak progress step.

Recall that convergence considers a worst-case synthesizer, which only returns a program from \( M^* \) when \( \gamma(S_i) \subseteq M^* \). Theorem 5.3 implies that for such a synthesizer, if a synthesis session reaches an inevitable failure point, the session can either be infinite or end with \( q_n = \bot \). This means that backtracking is necessary. However, in the worst case the failure may become observable to the user only when (if) the session terminates with \( q_n = \bot \). A more sophisticated user may realize this earlier, at the first inevitable failure point where \( \gamma(S_i) \cap M^* = \emptyset \). We refer to this point as the first infeasible point, and to the prior point as the last feasible point. We note that these points are only observable if \( M^* = U^* \) (or if the user is aware of \( M^* \)).

### 5.5.1 Unbounded Backtracking

We now consider the amount of steps that have to be traced back from the point where \( q_n = \bot \) (i.e., the session terminates with failure) or from the point where \( \gamma(S_i) \cap M^* = \emptyset \) (i.e., the first infeasible point in the session) to recover a synthesizer state from which there is a suffix that leads to successful convergence. We argue that there is no bound on the number of steps that we need to backtrack; this is demonstrated via the following scenario.

Consider a syntesizer where \( M \) is all the programs in a language generated by \texttt{if} expressions, equality (\texttt{==}), all list constants over integers (e.g. \texttt{[]} , \texttt{[1, 2, 3]} etc.), recursive call \texttt{f}, the input variable \texttt{i}, and the library functions \texttt{cons, max, remove, sort, and reverse}.

The predicate set \( \mathcal{P} \) contains all input-output examples \((i, \omega)\), and syntactic exclusion of a single element, that is “exclude \( e \)” for \( e \in \{ \texttt{if, ==, cons, } \cdots \} \).

The user wants to sort a list of integers in descending order. The following table shows a possible interactive session with the synthesizer.
The first two examples lead the synthesizer to generate a simple list reversal program. The user is not interested in this program, and disqualifies it by excluding \texttt{reverse}. The synthesizer then, quite unfortunately, takes the path of over-fitting the example set via branching using the \texttt{if} construct with equality conditions. The user keeps providing examples, but is handed an ever-growing chain of programs. After \(n\) such steps, the user chooses to block the synthesizer from over-fitting to particular inputs by excluding the equality operator, at which point the synthesizer can no longer find a program in \(M\) that satisfies \(S_{n-1}\), and \textit{Select} returns \(\bot\).

**Core set**  The core set \(B\) for this instance is the set of all finite sets of predicates containing no contradiction and (at least)

- One of \{\texttt{exclude if}, \texttt{exclude ==}\}
- Two examples \{\((\iota_1, \omega_1), (\iota_2, \omega_2)\)\} with \(\iota_{1,2}\) two lists such that \(|\{x \in \iota_1 \mid x > \text{head}(\iota_1)\}| > |\iota_2|\), and \(\omega_{1,2}\) their corresponding descending sorts.

To see why this is the core set, first note that the exclusion of either \texttt{if} or \texttt{==}, rules out conditionals as well as any form of recursion (since any recursive call will then be infinite). Including two input examples with the specified property rules out programs that use \texttt{remove} to reorder the elements.\(^1\) Moreover, when excluding neither \texttt{if} nor \texttt{==}, no number of examples is sufficient to make a complete specification since switch-like over-fitting is always a valid solution.

**Inevitable failure point**  In this example, an inevitable failure point occurs after the second step. The reason being, that any \(m \in \gamma(B)\) must use \texttt{reverse}, since any

\(^1\) The number of \texttt{remove}s has to be at least \(|\{x \in \iota_1 \mid x > \text{head}(\iota_1)\}|\), but at most \(|\iota_2|\), which is not possible without branches.
non-recursive program without it can correctly order only a fixed number of elements from the input. \{exclude reverse\} disallows that, leading to $\gamma(B) \cap \gamma(S_1) = \emptyset$.

It is possible for a correct user to reach this state, since the user expects the program `sortBy(i, neg)` which is a valid program ($\in U$) — but this program is beyond the synthesizer’s search space ($\not\in M$).

**First infeasible point** It should also be noted that that after the second step, $\gamma(S_1) \cap M^* \neq \emptyset$, since if (i==[]) [] else cons(max(i), f(remove(i, max(i)))) (also known as max-sort) is a realization of the goal. So $S_1$ is still a feasible point, and so are $S_{2.,(n-2)}$ — since the examples consist of valid descending sorts, hence max-sort $\models A_{2..(n-2)}$. Max-sort is only discarded at $A_{n-1}$, by the exclusion of $==$, and since reverse has already been excluded, reverse(sort(i)) or any other composition of sort and reverse cannot be generated. Now, $\gamma(S_{n-1}) \cap M^* = \emptyset$, making iteration $n$ the first infeasible point. It so happens that the three examples shown are enough to make $\gamma(S_{n-1})$ empty, so the synthesizer returns $\bot$.

The last, important thing is that we can construct the session with an arbitrarily large $n$, such that the inevitable failure point ($i = 2$) is any number of steps away from the last feasible point ($i = n - 1$), and also from the actual failure with $\bot$ ($i = n$). It means that any bounded backtracking is insufficient for recovering the session in this case.

**Theorem 5.4.** For any given $k \in \mathbb{N}$, there exist:

1. a session $S$ of length $k + i$ where $S_i$ is an inevitable failure point and $q_{k+i} = \bot$.

2. a session $S$ where $S_i$ is an inevitable failure point and $S_{k+i}$ is the first infeasible point.

**Proof.** Using the construction described above, having $i = 1$ and either $n = k + 1$ (for 1) or $n = k + 2$ (for 2). Notice that in this scenario, the $n$th iteration exhibits both a first infeasible point and failure with $\bot$. \qed
Chapter 6

Recommendations for Future Synthesizers

In this chapter we draw conclusions from the work presented thus far, and discuss the implication of empirical results shown and of the conditions posed in definitions and theorems in the previous chapters.

6.1 Progress models

Progress of the synthesizer is important not only for making sure the session will converge, but also as a tool for the user to understand their status in the synthesis session.

Synthesizers that do not actively define themselves as iterative have no way of enforcing progress, of course, but if the implementation of Select is order-dependent, then the user can tell whether their feedback has moved the session along. This is tricky when considering weak progress—Select might stop at the same program even though other programs have been eliminated from the space. This, of course, is the danger of weak progress. One of the reasons it would be helpful for synthesizers to start considering themselves as interactive is so they can provide this feedback to the user, and limit frustration and confusion.

We have already seen an example of a synthesizer that enforces very strict strong progress in FlashFill, and FlashExtract [LG14] and BlinkFill [Sin16] follow the same workflow. GIM (Chapter 4) puts forth a set of predicates that allow the user to provide positive feedback on the program, which means that even if strong progress is to be enforced, it must be enforced at the more relaxed level described in definition 5.3.10, allowing predicates that hold for the current program along with those that rule it out. In an enumerating synthesizer that unifies sub-programs based on observational equivalence, such as [OZ15], weak progress may be sufficient: a change in the search space could change the equivalence classes created while enumerating, leading to a different result from Select even though the current program was not eliminated. This
could also aid a realistic user who might not be completely certain whether a program is in $M^*$.

When designing a new synthesizer, there are pros and cons to each of the progress models. Strong progress, paired with a Select that will return the same program again and again, will reduce user frustration. Weak progress has been shown in Chapter 4 to help an uncertain user reach a better program. However, the feasibility of enforcing the progress model is itself an issue: strong progress is easy to test, as it only requires for the user answer to rule out the current program. Weak progress, as seen in lemma 5.3.8, requires the ability to check implication of the predicates over the current domain of programs. This, even for simple predicates, may be difficult.

There is also the possibility of not enforcing progress at all. It can be easily seen that termination, as proved in section 5.4, is not impeded if the user provides finitely many answers that do not make progress along with those that do. (This also applies to finitely many steps made by predicates for which termination is not guaranteed). However, we believe forcing progress is a way to keep the user on track.

### 6.2 Realizability gap

One of the problems a synthesizer can suffer from is a gap between the expectations of the user and the ability of the synthesizer. Often, this is expressed by the fact that $M^* \subset U^*$, as in the example in section 5.5.1. In such a case, a user can repeatedly backtrack and try new predicates, and still fail because they may not even be able to pinpoint the first infeasible point of a session, let alone the initial point of inevitable failure of their session.

Unfortunately, there is not much that can be done about this, especially since limitations on the expressibility of $M$ have been previously shown to be important for both termination [LMN16] and for heuristically arriving at the user’s intentions faster [LG14]. All that remains for the synthesizer to do is to better communicate the limitations of $M$.

### 6.3 Sharing more with the user

Section 4.5.5 includes the surprising result that users are not as correct without the additional information provided by user-provided inputs and intermediate results executing them. In an interactive session, where the user is constantly asked to examine programs, it is important that the synthesizer help communicate to the user as much about a candidate program as possible. This added information can be debug information, as in Section 4.4, types, access to documentation, and more.

One of the design tenets behind the Granular Interaction Model is to enrich the interaction model with the user and to include more information about the program. Another way in which the interaction can be made more informative is by communicating
more information about the state of the synthesizer. The suggestion in Section 6.1 of an indication of whether, and what level, of progress has been made is an example of this.

Similarly, the synthesizer can communicate additional data about $M$ and $\mathcal{P}$. Showing the user a visualization of the remaining search space may help with problems such as the realizability gap or to identify points of failure faster. Suggesting to the user stronger predicates they may wish to use in their answer might help the process terminate faster.
Chapter 7

Related Work

Enumerative syntax-guided synthesis (SyGuS) [ABJ+15] is the domain of program synthesis where the target program is derived from a target programming language according to the syntax rules. [ISSL+16b, LWDW01, URD+13, RVY14] all fall within this scope. The implementation of GIM presented in this thesis is syntax-based, where the target language is a functional subset of Scala as specified by $\mathcal{V}$. Syntax-based synthesis algorithms often use a user-driven interaction model [Gul12], which GIM extends.

Counterexample-guided inductive synthesis (CEGIS) is a synthesis framework that has been formalized in [SL08] and [LPP+17]. It is implemented in tools such as Sketch [SLTB+06, SLJB08], which allows the user to restrict the search space via structural elements (e.g. conditions or loops) containing holes to be synthesized. Sketching is a way to leverage a programmer’s knowledge of expected syntactic elements, and when used in conjunction with restrictions on the syntax [ABJ+15] can allow very intricate synthesis. Sketch exhibits two forms of iterative processes: the first one is an internal loop that involves a solver and a verifier, where the solver attempts to fill the holes in the sketch and the verifier provides a stream of input-output examples until the result passes validation; and the second, external one involves the human user and the tool, where the user may not like the generated program or the tool rejects the sketch because it is unsatisfiable. The internal loop is example-driven, with the verifier taking the place of the user. The external one is non-monotonic, as the user can remove assertions from the specification or change the syntactic class of the program entirely. The only monotonic changes are (i) adding an assertion, (ii) removing an assumption, and (iii) replacing a numeric hole with a constant.

Type-directed synthesis is a category of synthesis algorithms that perform syntax-based synthesis mainly driven by the types of variables and methods, and the construction of the program is performed through type-derivation rules. While type-directed methods tend to be user-driven, many of them [GKKP13, GRB+14, PGBG12] require only initial
specifications and the user manually chooses from multiple candidate programs that match the specification. The philosophy behind GIM is that a user should not consider many programs (there could be dozens or more) at a time with no additional data. Rather, programs should be considered one at a time, with additional information that can help the user consider the program in depth and direct the search.

Synquid [PKSL16] is a type-directed synthesis tool that uses refinement types, which encode constraints on the solution program to be imposed on the candidate space. Refinement types have rich semantics and a definition of subtyping based on logical implication. The user can add syntactic structure (roughly, the top of the tree) to help the synthesizer, and can also strengthen the return type of the program (by replacing it with a subtype) or loosen the precondition for the types of the arguments (by replacing them with a supertype). These are all monotonic progression steps, but the user can also change a type to any other type or change the number of inputs to the program, which are not monotonic. Tools that combine type-directed synthesis with examples [OZ15, FCD15, FMW+17a] make for a more iterative model, as adding examples is always monotonic.

**Formal models of synthesis procedures** Models of families of synthesizers exist for enumerative, syntax-based synthesizers [ABJ+15], VSA-based synthesizers [PG15], and oracle-driven synthesizers via inductive learning [JS17]. These all describe a single-iteration interaction with the user (though [JS17], which describes the counterexample-driven model as well, does describe iterative behavior with the oracle). Two recent works describe an iterative model of interactive synthesis. One [LPP+17] focuses on the synthesizer-driven model of interactive synthesis: the synthesizer asking the user about differentiating examples, and turning the answer back into constraints on the search space. This model is somewhat specialized for VSA-based synthesizers and is an interactive expansion of [PG15]. The work of Loding et al. [LMN16] which is intended mostly to describe the internal iteration of a CEGIS synthesizer, is also suited to a user-driven model of interactive synthesis, as is the one presented in Chapter 5. The model is based in machine learning terminology, with a teacher-learner model exploring a hypothesis space (i.e., a space of programs or other classifiers), and use a sample space containing input-output examples and no additional forms of feedback. Finally, they offer a weaker termination result, showing the existence of a terminating learner (user) hinging on an ordering of the hypothesis space.

**Programming by Example** In PBE the interaction between user and synthesizer is restricted to examples, both in initial specifications and any refinement. FlashFill [Gul11] is a PBE tool for automating transformations on an Excel data set, and is included in Microsoft Excel. It does not show its users the program, only its application on the data set. This means that the result of the synthesis is not reusable to any other data set since it might hide unintended behavior that happens to work on the current data set.
The FlashMeta [Sin16, PG15] Because the resulting program is never inspected, it might still suffer from overfitting to the examples and is not reusable. Escher [AGK13] is a PBE tool for synthesizing recursive functions. Like FlashFill, Escher decomposes the task based on the examples, searching for programs that could be used as sub-programs in condition blocks. Escher is parameterized by the operations used in synthesis, and like FlashFill, allows refinement only by re-running the process.

Adding examples to Type-directed synthesis Recent work connects PBE with type-driven synthesis [OZ15, FCD15]. These tools accept examples (and their inherent type information) as initial specifications, use type derivations to produce candidates, and verify them with the examples. Bigλ [SA16] synthesizes MapReduce processes via sketching and type derivations over lambda calculus and a vocabulary. Examples are also used to verify determinism. SyPet [FMW+17a] is a type-directed, component-based synthesis algorithm that uses Petri-nets to represent type relationships, and finds possible programs by reachability. Candidates are tested using tests provided by the user. SyPet requires full test cases rather than examples, which, while more descriptive, still require the user to learn a lot about the library in order to program the test case, an effort that may be equal to learning about the methods required to solve the programming task at hand.

Sketching The user can restrict the search space via sketches [SL08, SLTB+06, SLJB08], structural elements (e.g. conditions or loops) which includes holes to be synthesized. Sketching is a way to leverage a programmer’s knowledge of expected syntactic elements, and when used in conjunction with restrictions on the syntax [ABJ+15] can allow very intricate synthesis. However, since the most general sketch, a program with only a single hole, is usually too unconstrained for the synthesizer, the user must come armed with at least some knowledge of the expected structure rather than iteratively build it as in GIM.

Enriching user input Several existing works have enriched the specification language, or the interface for specifying program behavior. Adding examples to type-directed synthesis is an example of such enrichment. Another approach by Polikarpova et al. [PKSL16] with SYNQUID is to use refinement types instead of types, which encode constraints on the solution program, which can be imposed on the candidate space. While these constraints are mainly semantic, unlike GIM’s syntactic predicates, this embodies the same ideal of passing off some responsibility to a user who can understand code, or in this case, write code. Likewise, Barman et al. [BBC+15] suggest an interactive, user-dependent extension of sketching intended to synthesize the sketch itself by leveraging the user to decompose the specifications and examine the results. Angelic programming [BCG+10] leverages programmer knowledge by an expanded interface from synthesizer to user: the user is shown a synthesized program with a nondeterministic
“angelic operation” and execution traces for that operation to make the program correct, and it is their responsibility to identify the necessary operation to replace the angelic operator.
Chapter 8

Conclusion

The goal of this thesis was to identify the points of intervention where the human user can assist the synthesizer, and to improve the interaction in a way that lets the user be of assistance to the synthesizer while not being overburdened.

In Chapter 3 we described JARVIS, a tool to extract repetitive tests from unit test suites and synthesize from them property-based tests. We have shown the foundations for its operation: sorting the existing unit tests into sets of compatible tests; a better abstraction, achieved using a hierarchy of generality between parameterized tests which allows abstractions to generalize more tests; generalizing the examples to a data generator, taking into consideration the positive and negative examples (tests expected to succeed and fail, resp.) using the notion of Safe Generalization; and preserving the subtleties of human-written unit tests by sampling each abstracted region according to constraints found in the test data.

We applied JARVIS to the JUnit test suites of 12 Apache Commons APIs, and have shown there is ample repetition in the data of real-world test suites, which can be used to generate PBTs. We have also shown that the repetition often includes subtleties, in the same testing scenario. Additionally, we have shown that JARVIS-generated PBTs maintain the instruction coverage of the original unit tests, and increase parameter value coverage by as much as two orders of magnitude. PBTs generated by JARVIS have found a known bug in Apache Commons-Math, and with the help of JARVIS we identified a bug in the Commons-Lang test suite.

JARVIS is not an iterative tool by design, like many synthesizers, and additionally cannot be run iteratively the way many synthesizers are, since re-running it requires writing new unit tests. Our benchmarks, which involved generalizing the unit tests of Apache Math into property-based tests, included an instance which required manual intervention by the user. In this, JARVIS is a demonstration of the power a programmer can bring to synthesis as its user.

Expanding on this insight of Chapter 3, Chapter 4 defines the granular interaction model (GIM) for interacting with a synthesizer. This interaction model extends common PBE approaches and enables a programmer to communicate more effectively with the
synthesizer.

We proved that using only examples is insufficient for eliminating certain undesired operations in a program, where these undesired operations are easy to eliminate when using syntactic operations made available by GIM.

We further showed the effectiveness of GIM by a controlled user study that compares GIM to standard PBE. Our study showed that participants have strong preference (66% of the time) for granular feedback instead of examples, and were able to provide granular feedback up to 3 times faster (and 2.14 times faster on average).

In Chapter 5 we formalized the sessions described in Chapter 4 and presented a model of iterative synthesis sessions that we used to define and prove properties of iterative synthesis. Using this model we described progress of a synthesis session and proved two conditions for termination of a session, one depending on the language of feedback predicates and one depending on the behavior of the user. Additionally, we proved that backtracking from a failed synthesis session is potentially unbounded.

Finally, in Chapter 6 we collected conclusions from the results in previous chapters into recommendations for future synthesizers.

Overall, the approaches described in this thesis demonstrate the power of allowing a power user such as a programmer to be a part of the synthesis loop. Our formal model points out important information that should be communicated to the user in an ideal synthesizer. GIM, our main solution, also reveals new challenges and ways in which this approach should be developed in the future.
Bibliography


[jac] Eclemma - jacoco java code coverage library.


[jsv] Jsverify: Write powerful and concise tests.


[scab] Scalagen: Java to scala conversion.


This page contains text in Hebrew, discussing the need for expressive support for developers, who can view the program created from the entire intended interaction with the program. In addition, the users can determine whether the program is correct and evaluate its success and execution method using the GIM. The authors argue that developers can use this system to model the interaction in a granular and dynamic way, allowing users to estimate its effectiveness and contribute to their work.

The model, which is formalized by the developers and integrated into the synthesizer during runtime in a general and formal manner, is used to calculate the area of the model for each iteration and its application within the program possibilities of the synthesizer. This model allows the enumeration of various conditions for the synthesis in the worst case, while also considering the characteristics of the language used by the user. Finally, the model shows the conditions for the execution of the user's constraints on the synthesis in the worst case, when the user fails to perform a task.

The model is supplemented and completed with other elements as shown, and it is clear that the model is effective in fulfilling its purpose.
Tакзер

The synthesis of functions is a complex and challenging task, especially for non-expert users. The generation of code, even for experienced programmers, can be a difficult and error-prone process. This is where automated code generation comes into play. By using algorithms, computer programs can generate code that meets specific requirements, often automatically, based on the user's inputs.

In recent years, there has been significant progress in the field of automated code generation. These techniques allow for the creation of code based on specified intentions, often in a convergent manner. However, the process is still dependent on the user's actions, and the generated code cannot always meet all the requirements.

The generated code is often presented in a way that is not directly usable, requiring further refinement by the user. This can be a challenging task, especially for non-expert users. Moreover, the generated code can be limited in scope, often only providing a subset of the desired functionality.

The thesis focuses on the design of a tool that can generate code based on user specifications, aiming to simplify the process and make it more accessible. The tool is designed to be able to handle a wide range of specifications, from simple to complex, and to provide a user-friendly interface.

The tool's main feature is its ability to generate code that meets the user's specifications, even in cases where traditional approaches fail. The generated code is presented in a way that is easily understandable and can be further refined by the user.

The thesis also discusses the challenges of automated code generation and the limitations of existing approaches. It proposes solutions to overcome these limitations and to make code generation a more accessible and user-friendly process.

The thesis contributes to the field of automated code generation by providing a new tool that can generate code based on user specifications, making it easier for users to create code without needing to have extensive programming knowledge.
המחקר בוצע בחנאותי של פרופסור ערן יב, בכפיפות ל umożliיחו.

חלק מההתוצאות בתוכנית הפרסום הופיעו במאת מחבר הגנרטיקטי של המחבר, אשר גרסאותיו העדכניות יוחנות מצוינת:


סינתזה של תכונהELCOME

היבר על מחקר

לשמ מילוי חלקי של הדרישות לקבלת התואר

דוקטור לפילוסופיה

הילה פלג

הילה פלג

הוג שאנ getDate - מכון טכנולוגי לישראל

אב התח"ש ת"ח היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ היפ HOP2019 2019
סינתזה של תכונה
עבורה מתכנתים

הילה פלג