Programming Not Only by Example

Hila Peleg
Technion
hilap@cs.technion.ac.il

Sharon Shoham
Tel Aviv University
sharon.shoham@gmail.com

Eran Yahav
Technion
yahave@cs.technion.ac.il

Abstract

Recent years have seen great progress in automated synthesis techniques that can automatically generate code based on some intent expressed by the programmer, 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 programmer to choose from, shifting the heavy-lifting to the user. An alternative approach, successfully used in end-user synthesis, is programming by example (PBE), where the user leverages examples to interactively and iteratively refine the intent. However, using only examples is not expressive enough for programmers, who can observe the generated program and refine the intent by directly relating to parts of the generated program.

We present a novel approach to interacting with a synthesizer using a granular interaction model. Our approach employs a rich interaction model where (i) the synthesizer decorates a candidate program with debug information that assists in understanding the program and identifying good or bad parts, and (ii) the user is allowed to provide feedback not only on the expected output of a program but also on the program itself. After identifying a program as (partially) correct or incorrect, the user can also explicitly indicate the good or bad parts, to allow the synthesizer to accept or discard parts of the program instead of discarding the program as a whole.

We show the value of our approach in a controlled user study. Our study shows that participants have a strong preference for granular feedback instead of examples and can provide granular feedback much faster.

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 [13] to formulating SQL queries [39]. Tools for expert users who can encode full specifications have also matured enough to be practical [23, 32, 36].

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 [7, 9, 16–18, 23, 26, 36–38], 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., [12, 15]). 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 [5, 13, 20–22, 24, 39–41] or as a way to refine the results of type-driven synthesis [11, 25].

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.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180189

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 paper 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.

Outline In Section 4 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 5 in addition to examples. In Section 5.2, we also introduce debug information for every example provided by the user. Section 6 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.

Task: find the most frequent bigram in a string

Initial example (σ_0)	`"abdfibfcfdebdfdebdihgfkjfdebd"↦"bd"`
Question q_1	`input` `.takeRight(2)`
Problem: `takeRight` will just take the right of a given string **Idea:** the frequent bigram needs to be placed in the middle	
Answer σ_1	`"cababc"↦"ab"`

Question q_2	`input`
	`.drop(1)`
	`.take(2)`

Problem: this program crops a given input at a constant position **Idea:** vary the position of the frequent bigram between examples	
Answer σ_2	`"bcaaab"↦"aa"`

Question q_3	`input`
	`.zip(input.tail)`
	`.drop(1)`
	`.map(p => p._1.toString + p._2)`
	`.min`

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 σ_3	`"xyzzzy"↦"zz"`

Table 1: The difficulty of finding a differentiating example.

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 $\mathcal{V}$. 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:

$$\sigma_0 = \texttt{"a}\underline{\texttt{bd}}\texttt{fibfcfde}\underline{\texttt{bd}}\texttt{fde}\underline{\texttt{bd}}\texttt{ihgfkjfde}\underline{\texttt{bd}}\texttt{"} \mapsto \texttt{"bd"}.$$

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

2.1 Interaction with a classic PBE synthesizer

Table 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 σ_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 σ_0 and σ_1.

The interaction proceeds in a similar manner. Each additional example may reduce the number of candidate programs (as they are

Task: find the most frequent bigram in a string

Initial specifications		`"abdfibfcfdebdfdebdihgfkjfdebd"`↦`"bd"`
Question q_1	1 2	`input//"abdfibfcfdebdfdebdihgfkjfdebd"` `.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 q_2	1
	2
	3
		`input //"abdfibfcfdebdfdebdihgfkjfdebd"`
		`.drop(1)//"bdfibfcfdebdfdebdihgfkjfdebd"`
		`.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 q_3	1
	2
	3
	4
	5
		`input //"abdfibfcfdebdfdebdihgfkjfdebd"`
		`.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 2: Providing granular, syntactic feedback.

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 `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 `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 σ_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 `min` requires deep understanding of the program, which is then only leveraged implicitly (through examples). Even then, there is no guarantee `min` will not recur: as we will show in Section 4, it cannot be removed completely in this model. In this case, since the programmer already knows that programs using `min` should be avoided, this information is best communicated this information explicitly to the synthesizer.

2.2 Interaction through a granular interaction model

GIM improves PBE by employing a richer, *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 2. Question 1 is as before: the synthesizer produces the candidate program `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 q_1, this is just the input and output values of the initial example. In the next steps this information will be far more valuable.

Given q_1, the programmer responds by providing *granular feedback*. Using GIM it is possible to narrow the space of programs using syntactic operations. Presented with `input.takeRight(2)`, the user can *exclude* a sequence of operations from the vocabulary, in this instance `takeRight(2)`, ruling out *any program* where `takeRight(2)` appears. This also significantly reduces the space of candidate programs considered by the synthesizer.

The synthesizer responds with q_2. 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 q_2, the user rules out the sequence `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 q_3. This candidate program contains something the programmer would like to preserve: the debug information shows that the prefix `input.zip(input.tail)` creates all bigrams in the string. The user can mark this prefix to *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 `take(2)` since the resulting truncation of the list is undesirable.

Eventually, the synthesizer produces the following program:

```
input//"abdfibfcfdebdfdebdihgfkjfdebd"
.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","ib",...)
.groupBy(x => x)//Map("bf"->List("bf"),"ib"->List("ib"),...)
.map(kv => kv._1 -> kv._2.length)//Map("bf"->1,"ib"->1,"gf"->1,...)
.maxBy(_._2)//("bd",4)
._1//"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.

3 Background

In this work we address synthesis of functional programs. Below we provide the necessary background.

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, $[\![m]\!] : D \to D \cup \{\bot\}$ 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) $\bot \notin D$.

Vocabulary and the candidate program space The candidate program space consists of programs of the form `input.f1.....fn-1.fn` (in Scala notation), or $f_n(f_{n-1}(\dots f_1(input)\dots))$ (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.

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 $(\mathcal{E}, \mathcal{L})$ of initial examples $\mathcal{E}$ and target language $\mathcal{L}$, where each example in $\mathcal{E}$ is a pair (i, o) of input $i \in D$ and expected output $o \in D$. The result of the PBE problem $(\mathcal{E}, \mathcal{L})$ is a program m, which is a valid program in $\mathcal{L}$ that satisfies every example in $\mathcal{E}$, i.e., $[\![m]\!](i) = o$ for every $(i, o) \in \mathcal{E}$. Since there might be more than one program m in the language $\mathcal{L}$ 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 $\mathcal{E}_i$ that direct it in continuing the search.

4 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 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 2.1, the user wishes to rule out the function `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 $[\![m_1]\!] = [\![m_2]\!]$. We use this to prove the following claim:

CLAIM 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 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 consider two classes of methods: 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 `in.reverse.reverse` will be identical to `in`. An invertible method can always be added to the target program along with its inverse, resulting in an equivalent program. Hence, it will never be ruled out by examples.

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.

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.

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

The practical implications of claim 1 are discussed in Section 6.4, 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 6).

5 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.

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

(1) *remove*$(f_i, \ldots, f_j)$ where $i \leq j$: will hold only for programs m' where $\neg\exists k. f'_k = f_i \wedge \cdots \wedge f'_{k+i-j} = f_j$
(2) *retain*$(f_i, \ldots, f_j)$ where $i \leq j$: will hold only for programs m' where $\exists k. f'_k = f_i \wedge \cdots \wedge f'_{k+i-j} = f_j$
(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 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 5.4 will expand on the reasons to select certain expansions to the interaction model over others.

5.2 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 $\mathcal{E}$. In a functional concatenation, it is possible to show the user the result of each subprogram, on each $e \in \mathcal{E}$. This means that even for some

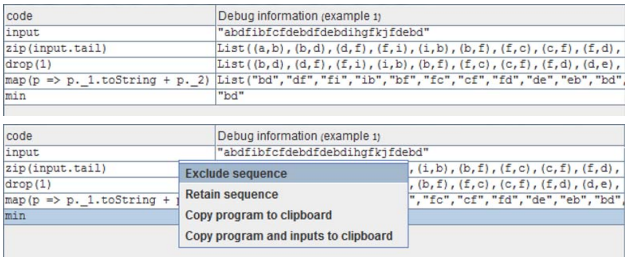

Figure 1: Program with debug information, a sequence selected for removal

unfamiliar $f \in \mathcal{V}$, the user can still gauge its effect and determine *by example* whether that effect is desired.

EXAMPLE 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:

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

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.*

5.3 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 6.3.

A candidate program is displayed to the user alongside the debug information. The top image in fig. 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. 1.

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

5.4 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 5.1 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 [3, 4, 11]. For the domain considered in this paper, 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 $\mathcal{V}$. 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 $f \in \mathcal{V}$ 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.

6 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 5, 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 5.1 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 5.4 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 6.2, 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 6.3, it also precomputes the space of valid programs.

6.1 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 Tab. 3, were each paired with a vocabulary and an initial set of examples.

Discussion As seen in Tab. 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.

6.2 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 paper), making an effort to create a run with fewer iterations and more aggressive pruning of the space in each iteration.

Results Tab. 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/.

6.3 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?*

	Benchmark		$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}_{init}\rvert$	$\lvert m^*\rvert$	candidate space size	Number of Candidates: reject only	PBE	Syntax	GIM
strings	dropnthletter	Drop every 5th letter in a string	20	1	3	280	4	2	3	2 (1)
	freqbigram	Most frequent bigram in a string	19	1	6	118261	674	4	8	6 (0)
	frequword	Most frequent word in a string	25	1	4	4853	126	5	8	6 (1)
	linesinfile	Number of lines in file	20	1	2	47	4	3	4	3 (1)
	nonemptylines	Number of non-empty lines in file	21	1	3	1664	29	2	3	3 (1)
lists	anagrams	Group words that are anagrams	17	1	6	13554	12	3	3	3 (0)
	histogram	Create a histogram of number list	12	1	5	4208	37	3	9	3 (1)
	median	Find the median of a list of numbers	20	1	4	71211	1663	6	14	9 (1)
	posinlist	Get all positive numbers from list	20	1	2	190	17	3	4	4 (1)
	sudokusquare	Validate a square in sudoku	17	1	5	1602	118	3	7	4 (0)
	sumsquares	Sum of squares of a list of numbers	20	1	2	120	2	2	2	2 (0)
streams	bitstream	Next integer from a stream of bits	17	2	4	3717	101	2	8	5 (1)
	numhashtags	Count hashtags in a stream of tweets	15	1	7	11527	2	2	2	2 (0)
	slidingavg	Average next three values from every index	25	1	4	60479	125	2	2	2 (0)

Table 3: The test setup of 14 synthesis experiments, showing the ambiguity inherent in $\mathcal{E}_{init}$, and the number of iterations to the target program in an ideal synthesis session with each available set of operations. Parentheses indicate examples used.

(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 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 6.2 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.

task	group	no. of sessions	iteration time (sec) avg	iteration time (sec) med	number of iterations avg	number of iterations med	finished	correct: target answer	correct: equiv. answer
histogram	PBE	11	163.34	131.97	2.45	2.0	11	1	10
	Syntax	9	86.27	59.97	12.11	8.0	9	3	4
	GIM	10	98.78	96.18	8.90	7.5	10	5	4
no. lines with text	PBE	11	170.13	168.17	2.73	2.0	11	0	8
	Syntax	8	82.16	60.56	10.50	9.5	7	3	1
	GIM	11	78.26	65.78	8.82	8.0	9	4	3
most frequent word	PBE	11	114.52	71.27	4.45	4.0	10	9	0
	Syntax	10	58.28	50.34	22.10	15.0	8	7	0
	GIM	11	79.87	53.84	8.82	8.0	10	8	0

Table 4: Summary of the three tasks performed in the user study (all users).

Implementation Participants performed the tasks using the UI shown in Figure 1. The space of programs was precomputed by the enumerating synthesizer from Section 6.1 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.

6.4 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. Medians are also shown in Fig. 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 frequword (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 frequword (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=0.047), but not for nonemptylines (60.56s, 65.78s, p=0.33) or frequword (50.34s, 53.84s,p=0.19).

	task	no. of sessions	sessions used examples	percent examples per user avg	med	min	max
all users	histogram	10	9	37.6%	35.0%	0.0%	85.7%
	nonemptylines	11	8	29.7%	31.0%	0.0%	66.7%
	frequword	11	10	36.1%	37.5%	0.0%	85.7%
users familiar with Scala	histogram	2	2	74.1%	74.1%	62.5%	85.7%
	nonemptylines	2	2	46.7%	46.7%	33.3%	60.0%
	frequword	2	2	29.9%	29.9%	22.2%	37.5%
users unfamiliar with Scala	histogram	8	7	28.5%	17.7%	0.0%	66.7%
	nonemptylines	9	6	25.9%	30.0%	0.0%	66.7%
	frequword	9	8	37.5%	37.5%	0.0%	85.7%

Table 5: Proportional part (%) of examples in the predicates provided by GIM group users. Some used no examples at all, none used only examples.

These results imply that, with the exception of the frequword 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 are visualized in Fig. 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 5 and Fig. 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.

	removed sequence	GIM/Syntax users saw and removed	PBE times seen in session min	max	distracting occurrences (average)	users distracted
num of lines	tail	84.2% (16)	1	4	2.8	45.5% (5)
	takeWhile(c => c != "\n")	73.7% (14)	1	4	2.7	54.5% (6)
	filterNot(c => c=='\r' || c=='\n')	57.9% (11)	0	3	2.3	27.3% (3)
	filter(!_.isEmpty)	27.3% (3)	0	3	2.3	27.3% (3)
	tail.takeWhile(c => c != "\n")	15.8% (3)	1	4	2.8	45.5% (5)
most frequent word	takeRight(1)	100.0% (12)	0	1	0	0.0% (0)
	drop(10)	84.6% (11)	0	1	0	0.0% (0)
	drop(1)	76.5% (13)	0	3	2.5	16.7% (2)
	takeRight(6)	76.2% (16)	1	3	2.4	58.3% (7)
	dropRight(1)	71.4% (15)	0	7	3.3	33.3% (4)
	take(5)	57.1% (12)	1	5	2.8	66.7% (8)
	last	42.9% (6)	0	3	3	16.7% (2)
	drop(10).drop(1)	41.7% (5)	0	1	0	0.0% (0)
	takeRight(6).takeRight(6)	38.1% (8)	1	2	2	8.3% (1)
histogram	toMap	57.9% (11)	1	1	0	0.0% (0)
	map(_._1 -> 1)	42.1% (8)	1	1	0	0.0% (0)
	zipWithIndex	26.3% (5)	1	1	0	0.0% (0)
	map(_._1.toInt)	15.8% (3)	1	1	0	0.0% (0)

Table 6: Frequently removed method sequences in the Syntax and GIM groups and their occurrence in the PBE group.

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 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 1 is neither a purely theoretical problem, nor a problem leading only to equivalent rather than correct programs, as seen in Table 4, but a real distraction from the ability to synthesize over an expressive vocabulary.

Figure 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 => 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

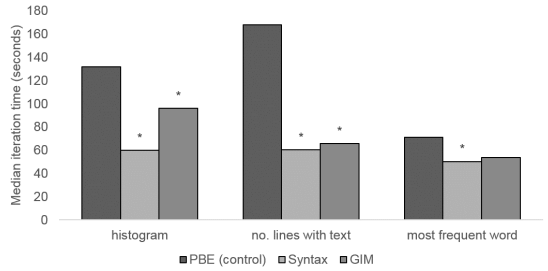

Figure 2: Median iteration time per task in each test group. Significant change from PBE is indicated by *.

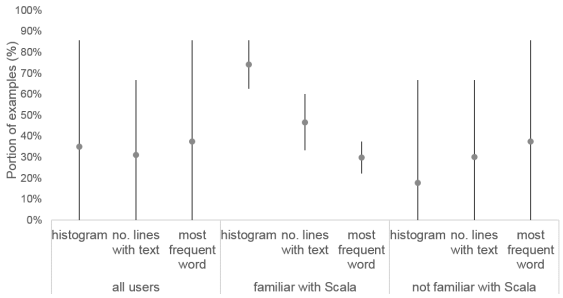

Figure 3: Examples used (med, min, max) by GIM users (all operations). None used 100% examples.

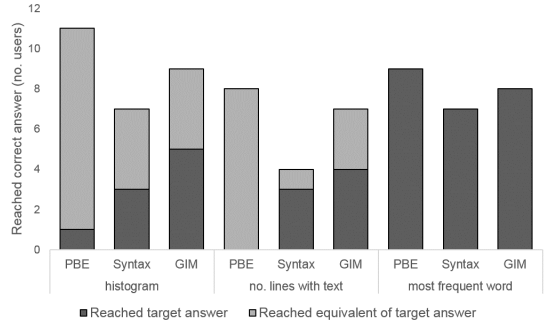

Figure 4: No. users that reached the target program or an equivalent (non-functional differences) result.

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/.

6.5 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 this difference between the two tests to the fact that the users in the GIM group resort to examples only when examples are convenient or readily apparent and therefore take less time to create.

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.

7 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.

Sampling of population An external validity issue mentioned in Section 6.5 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.

8 Related Work

Syntax-based synthesis [2] is the domain of program synthesis where the target program is derived from a target programming language according to the syntax rules. [19, 21, 30, 35] all fall within this scope. The implementation of GIM presented in this paper 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* [14], which GIM extends.

Programming by Example In PBE the interaction between user and synthesizer is restricted to examples, both in initial specifications and any refinement. FlashFill [13, 29] 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. Because the resulting program is never inspected, it might still suffer from overfitting to the examples and is not reusable. Escher [1] 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.

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 [12, 15, 27] 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.

Adding examples to Type-directed synthesis Recent work connects PBE with type-driven synthesis [11, 25]. 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 [31] synthesizes MapReduce processes via sketching and type derivations over lambda calculus and a vocabulary. Examples are also used to verify determinism. SYPET [10] 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 [32–34], 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 [2] 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. [28] 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. [6] 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 [8] 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.

9 Conclusion

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

Acknowledgements

The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC- COG-PRIME, as well as from Len Blavatnik and the Blavatnik Family foundation.

The authors thank Yifat Chen Solomon for her indispensable assistance in getting the user study off the ground, and Hadas E. Sloin and Yoav Goldberg for their help in analyzing the data.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International Conference on Computer Aided Verification*. Springer, 934–950.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

[3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. SyGuS-Comp 2016: Results and Analysis. *arXiv preprint arXiv:1611.07627* (2016).

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.

[5] Tobias Anton. 2005. XPath-Wrapper Induction by generalizing tree traversal patterns. In *Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken*. 126–133.

[6] Shaon Barman, Rastislav Bodik, Satish Chandra, Emina Torlak, Arka Bhattacharya, and David Culler. 2015. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 121–136.

[7] L. Blaine, L. Gilham, J. Liu, D. Smith, and S. Westfold. 1998. Planware - Domain-Specific Synthesis of High-Performance Schedulers. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE '98)*. IEEE Computer Society, Washington, DC, USA, 270–. http://dl.acm.org/citation.cfm?id=521138.786844

[8] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *ACM Sigplan Notices*, Vol. 45. ACM, 339–352.

[9] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C. Kuszmaul, Charles E. Leiserson, Armando Solar-Lezama, and Yuan Tang. 2016. AUTOGEN: Automatic Discovery of Cache-oblivious Parallel Recursive Algorithms for Solving Dynamic Programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 10, 12 pages. DOI: http://dx.doi.org/10.1145/2851141.2851167

[10] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2017*.

[11] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.

[12] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 653–663.

[13] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. DOI: http://dx.doi.org/10.1145/1926385.1926423

[14] Sumit Gulwani. 2012. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*. IEEE, 8–14.

[15] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 27–38.

[16] Daqing Hou and David M Pletcher. 2011. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 233–242.

[17] Shachar Itzhaki, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving Divide-and-Conquer Dynamic Programming Algorithms using Solver-Aided Transformations. In *international conference companion on Object oriented programming systems languages and applications*. To appear.

[18] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. 2010. A Simple Inductive Synthesis Methodology and Its Applications. *SIGPLAN Not.* 45, 10 (Oct. 2010), 36–46. DOI: http://dx.doi.org/10.1145/1932682.1869463

[19] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 145–164.

[20] Jürgen Landauer and Masahito Hirakawa. 1995. Visual AWK: A Model for Text Processing by Demonstration.. In *vl*. 267–274.

[21] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2001. Learning repetitive text-editing procedures with SMARTedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software* (2001), 209–226.

[22] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 55. DOI: http://dx.doi.org/10.1145/2594291.2594333

[23] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. 2012. Dynamic synthesis for relaxed memory models. *ACM SIGPLAN Notices* 47, 6 (2012), 429–440.

[24] Adi Omari, Sharon Shoham, and Eran Yahav. 2016. Cross-supervised synthesis of web-crawlers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 368–379. DOI: http://dx.doi.org/10.1145/2884781.2884842

[25] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.

[26] Robert Paige. 1990. Symbolic finite differencing-part I. In *European Symposium on Programming*. Springer, 36–56.

[27] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 275–286.

[28] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 522–538.

[29] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

[30] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 419–428.

[31] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 326–340.

[32] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. ProQuest.

[33] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 136–148.

[34] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 404–415.

[35] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[36] Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-Guided Synthesis of Synchronization. In ***POPL'10**: 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 327–338.

[37] Martin T Vechev, Eran Yahav, and David F Bacon. 2006. Correctness-preserving derivation of concurrent garbage collection algorithms. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 341–353.

[38] Martin T Vechev, Eran Yahav, David F Bacon, and Noam Rinetzky. 2007. CGC-Explorer: a semi-automated search procedure for provably correct concurrent collectors. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 456–467.

[39] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466.

[40] Ian H Witten and Dan Mo. 1993. TELS: Learning text editing tasks from examples. In *Watch what I do*. MIT Press, 183–203.

[41] Shanchan Wu, Jerry Liu, and Jian Fan. 2015. Automatic Web Content Extraction by Combination of Learning and Grouping. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1264–1274.