Abstraction-Based Interaction Model for Synthesis

Hila Peleg[1], Shachar Itzhaky[1], and Sharon Shoham[2]

[1] Technion, {hilap,shachari}@cs.technion.ac.il
[2] Tel Aviv University, sharon.shoham@gmail.com

Abstract. Program synthesis is the problem of computing from a specification a program that implements it. New and popular variations on the synthesis problem accept specifications in formats that are easier for the human synthesis user to provide: input-output example pairs, type information, and partial logical specifications. These are all partial specification formats, encoding only a fraction of the expected behavior of the program, leaving many matching programs. This transition into partial specification also changes the mode of work for the user, who now provides additional specifications as long as they are unhappy with the result. This turns synthesis into an iterative, interactive process.

We present a formal model for interactive synthesis, leveraging an abstract domain of predicates on programs in order to describe the iterative refinement of the specifications and reduction of the candidate program space. We use this model to describe the behavior of several real-world synthesizers. Additionally, we 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 show conditions for realizability of the user's intent, and show the limitations of backtracking when it is apparent a session will fail.

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 [10] to formulating SQL queries [34]. Programming by Example tools that accept input-output pairs as their specification have also matured enough to be practical on their own [2, 10, 17, 18, 20, 23, 34, 35, 36] or as a way to refine the results of type-driven synthesis [7, 24].

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 [11] 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) [33], 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. For instance, an input-output pair (i, o), often seen as the simple and natural tool for partial specification, can be seen as the predicate $[\![program]\!](i) = o$. As previous work [4, 26] has shown, examples are a weak tool with which to provide specification. The addition of other predicates in works such as [1, 24, 26] 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" [15] — by incrementally providing additional predicates to refine the specification, a process which at the limit will reach the comprehensive specification. In this paper, 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.

Goal The goal of this work 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 [3]. 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.

Existing work Previous work has modeled single iterations of different flavors of synthesis [1, 29], and the counterexample-guided model of synthesis (CEGIS) [14, 31]. The synthesizer-driven model of program synthesis [21] 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 [22] 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.

1.1 Our approach

In this paper, 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 $\mathcal{P}$, 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 $(2^{\mathcal{P}}, \supseteq)$, 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 3 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 specification 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 4 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 6 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 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 Finally, section 7 discusses the implications of these properties for the designers of future synthesis tools.

1.2 Main contributions

The main contributions of this paper 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.

2 Background

In this work we address program synthesis. Below we provide some background on synthesis methods and terminology.

The synthesis problem description Readers familiar with software verification would most likely recognize the common verification problem $\forall\iota.\ \varphi(\iota)$, where ι ranges over possible program inputs and φ 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 *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 φ is computationally hard.

Program semantics In the most abstract sense, a program m accepts input ι and produces output ω. 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 $[\![m]\!] : I \to O \cup \{\bot\}$. The special value $\bot$ indicates abnormal behavior, which may be a run-time error 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 φ 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)** is a sub-class of the synthesis problem where all communication with the synthesizer is done using input-output examples. The classic PBE problem is defined as a set $\mathcal{E}$ of examples, each of which is a pair of an input and its corresponding expected output; the result is a program m, within the candidate space, that satisfies every example in $\mathcal{E}$. PBE has become widely popular since examples are easier to create than full logical specifications, can be provided in many formats from tabular data to unit tests, and can even be created by non-programmers. 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, which 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.

  As an extension of this approach, *abstract examples* can be used to describe a (possibly infinite) set of examples using a short description. This description usually uses a weak abstraction mechanism, such as regular expressions.

3 Foundations for Synthesis with Abstract Domains

In this paper, 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 U 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 paper 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 $\mathcal{P}$ of predicates over programs. We assume that every $p \in \mathcal{P}$ is decidable. When considered against some set of programs T, each predicate $p \in \mathcal{P}$ 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 $\mathcal{P}$ 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 paper.

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 paper assumes working with a specific $\mathcal{P}$ 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 $\mathcal{P}$, and when used also in M^*.

3.1 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 $\mathcal{P}$. Formally, let $\mathcal{A} = 2^{\mathcal{P}}$. The synthesis process uses an abstract domain that consists of the powerset lattice $(\mathcal{A}, \sqsubseteq)$, where $\sqsubseteq$ 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 $A_1, A_2 \in \mathcal{A}$, meet is defined as $A_1 \sqcap A_2 = A_1 \cup A_2$ and join as $A_1 \sqcup A_2 = A_1 \cap A_2$. Further, $\top = \emptyset$ and $\bot = \mathcal{P}$.

From here on, we refer to $A \in \mathcal{P}$ 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).

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 $(2^M, \subseteq)$ and $(\mathcal{A}, \sqsubseteq)$.

Definition 1 (Abstraction). *For a single program $m \in M$, we define the* abstraction *function $\beta(m) = \{p \in \mathcal{P} \mid m \vDash p\}$, 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 $\alpha(C) = \bigsqcup_{m \in C} \beta(m) = \{p \in \mathcal{P} \mid \forall m \in C.\, m \vDash p\}$.*

This is similar to the abstraction performed by Houdini [8], Daikon [5], and D^3 [25].

Definition 2 (Concretization). *For an abstract element $A \in \mathcal{A}$, we define the* concretization *function $\gamma(A) = \{m \in M \mid \forall p \in A.\ m \vDash p\}$, or all programs for which all constraints in A hold.*

It is easy to verify that (α, γ) is a Galois connection.

Recall that in the abstract domain, $\bot = \mathcal{P}$ 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, $\gamma(\bot)$ 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 $\gamma(\bot)$ 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 $Select : \mathcal{A} \to M \cup \{\bot\}$, the synthesizer's operation of finding such a program. $Select(A)$ amounts to picking a concrete element from $\gamma(A)$, or returning $\bot$ if no such element exists; hence, it can be understood as partially concretizing the abstract element. The implementation of $Select$ is dependent on the synthesis algorithm being used.

3.2 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 $(name, \tau) \in Formals \times \mathcal{T}$ and a desired return-type predicate, $\tau_{ret} \in \mathcal{T}$ which will hold according to the $\sqsubseteq$ relation on types. A similar specification is used for type-directed synthesis that produces code snippets: the same τ_{ret} specifies the target type (usually assigned to a variable) and the available variables are specified using type predicates $(name, \tau) \in Vars \times \mathcal{T}$ for local variables $Vars$.

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, $[\![m]\!] : I \to O \cup \{\bot\}$, 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 ι, the program outputs ω. For this purpose, we define $m \vDash (\iota, \omega) \iff [\![m]\!](\iota) = \omega$.

Syntactic feedback as an abstract domain [26] introduces a domain of predicates that provide syntactic restrictions on programs, intended for use by programmers. For instance, an `include(f)` predicate which holds only for programs that make use of a function or operator f, or `exclude(f)` which holds only when they do not. For linear functional programs, these operators can also be generalized to sequences of methods, either as a continuous subsequence—`exclude(`$f{\cdot}g$`)` will hold only for programs where f is not immediately followed by g—or for general subsequences—`exclude(`$f{\cdot}g$`)` will hold for programs where there are no $i < j$ s.t. $f_i = f, f_j = g$. The predicates used in this work are limited to these, but in several examples in this paper we make use of predicates suggested by or simply in the spirit of those shown in [26].

3.3 Computability of the model

We notice that, in general, both α and γ are non computable: α because $\mathcal{P}$ may be infinite; and even though any A provided by the user will always be a finite set, γ 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 $Select(A)$ 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) [19], 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 [3], 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 $S^* \subseteq \mathcal{P}$ (or as full as $\mathcal{P}$ 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^*.

4 An Abstract Model of Interactive Synthesis

Section 3 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 3 (Synthesis session). *A* synthesis session *is a sequence of steps by the user and synthesizer $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ such that $q_i \in M \cup \{\bot\}$ are synthesizer questions and $A_i \in \mathbb{P}_{fin}(\mathcal{P}) \cup \{\bot\}$ are user answers, where $\mathbb{P}_{fin}(\mathcal{P})$ is the set of all finite subsets of $\mathcal{P}$ 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 4 (Synthesizer state). *The* state of the synthesizer *$S \subseteq \mathcal{P}$ 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 $A_0, A_1, \ldots, A_i \subseteq \mathcal{P}$, 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 $Select(S_{i-1})$, 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 $A_i \in \mathcal{A}$ in the form of an abstract element which consists of one or more predicates out of $\mathcal{P}$. Given the user's answer A_i, the new state of the synthesizer in step $i + 1$ is set to $S_{i+1} = S_i \sqcap A_{i+1}$, thus narrowing the set of concretizations to consider. Or, in other words, we can now define $S_{i+1} = \bigsqcap_{0 \leq j \leq i+1} A_j$. The search is over either when $Select$ 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 (User correctness). *A user step, providing A_i as an additional specification, is* correct *when $A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*.m \vDash p\}$.*

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, 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 6 (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 $\bot$.*
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 7 (Feasible synthesis session). *A* feasible synthesis session *is a synthesis session $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ that satisfies the following:*

(a) All A_i are correct steps (definition 5) or $\bot$,
(b) $q_i = Select(S_{i-1})$, i.e. $q_i \in \gamma(S_{i-1}) \cup \{\bot\}$, where $\bot$ signifies no possible program,
(c) If $q_n \in M^* \cup \{\bot\}$ then $\mathcal{S}$ is finite and of length n, and
(d) In a finite $\mathcal{S}$ of length n, $q_n \in M^* \cup \{\bot\}$

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 $Select$, if $\mathcal{S}$ 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 paper we are only interested in feasible synthesis sessions.

Definition 8 (Convergence). *A synthesis session* $(A_0, q1), (A_1, q_2), \ldots, (A_{n-1}, q_n)$ *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 7(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 $Select$, ranking, or domain knowledge, thereby causing the session to end immediately. Convergence, on the other hand, ensures that regardless of the implementation of $Select$, 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 *underapproximation* of the target. For convergence to be successful, that underapproximation must be nonempty. For the remainder of this paper we will be mostly interested in the worst-case implementation of $Select$, where the session either converges or is infinite.

4.1 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 will leverage progress into results on termination.

Definition 9 (Weak progress). *A user answer* A_i *is said to create* weak progress *in iteration* i *of a synthesis session if* $\gamma(S_{i-1} \sqcap A_i) \subsetneq \gamma(S_{i-1})$. *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 $S_{i-1} \sqcap A_i \sqsupset S_{i-1}$: 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, $S_{i-1} \sqcap A_i \sqsupset S_{i-1}$.

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

Proof. Let $\mathcal{S}$ be a synthesis session. Step i makes weak progress $\iff \gamma(S_{i-1} \sqcap A_i) \subsetneq \gamma(S_{i-1}) \iff \exists m \in M. m \in \gamma(S_{i-1}), m \notin \gamma(S_{i-1} \sqcap A_i) = \gamma(S_{i-1}) \cap \gamma(A_i) \iff \exists m \in M. m \in \gamma(S_{i-1}), m \notin \gamma(A_i) \iff \exists m \in M. \forall p \in S_{i-1}. m \vDash p, \exists p \in A_i. m \nvDash p \iff \exists p \in A_i. S_{i-1} \not\Rightarrow_M p \iff S_{i-1} \not\Rightarrow_M A_i$. □

Lemma 1 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 1. 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 10 (Strong progress). *A user answer A_i is said to create* strong progress *in the synthesis session if $q_i \notin \gamma(S_{i-1} \sqcap A_i)$, or in other words, if $\alpha(\{q_i\}) \not\sqsubseteq A_i$.*

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

Definition 10 is stronger than that defined in definition 9 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 2. The FlashFill implementation in Microsoft Excel [10] 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 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 11 (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 $diff(m_1, m_2) = \{p \in \mathcal{P} \mid m_2 \vDash p \wedge m_1 \nvDash p\} = \beta(m_2) \setminus \beta(m_1)$.*

Lemma 2 (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 $A_i \cap \bigcup_{m \in M^*} diff(q_i, m) \neq \emptyset$.*

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 12 (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., $\forall m \in U^*. m \nvDash p$.*

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 $(\perp_{\mathcal{P}}, \perp_M)$.

Example 3. Consider a domain of programs and a set of predicates $\mathcal{P} = \{\texttt{exclude(f)} \mid f \in \mathcal{V}\} \cup \{\texttt{include(f)} \mid f \in \mathcal{V}\}$ 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 $U^* = M^* = \{\texttt{input.tail.head}\}$, and that the user is shown $q_i =$`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, `exclude(head)` and `exclude(tail)`, will violate correctness, and will cause $S_i \cap M^* = \emptyset$.

5 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.

5.1 WQO predicates

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

Definition 13 (Well-quasi-order [16]). *Let $\leqq$ be quasi-order on X (i.e., $\leqq \subseteq X \times X$ is a reflexive and transitive relation). By convention, $x > y$ denotes $y \leqq x \wedge x \nleqq y$. The following definitions are equivalent:*

(1) $\leqq$ is a wqo over space X
(2) In every infinite sequence $x_1, x_2, \ldots$ there exist $i < j$ s.t. $x_i \leqq x_j$, 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 $x_i \nleqq x_j$ for $i \neq j$ is finite (the incomparable chain condition, also known as the antichain condition).

Theorem 1. *Let $p \preceq p' \iff p \Rightarrow_M p'$. If $\preceq$ is a well-quasi-ordering over the set $\bigcup_{m' \in M^*} \beta(m')$, 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 $\mathcal{S}$ 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 $\mathcal{S}$ makes weak progress, we know that $S_{i-1} \not\Rightarrow_M p_i$ (Lemma 1) and in particular, for every $p' \in S_{i-1}$, $p' \not\Rightarrow_M p_i$. From definition 4, $\forall p_j . j < i \Rightarrow (p_i \not\Rightarrow_M p_j)$, or in other words, $\forall p_j . j < i \Rightarrow (p_i \not\preceq p_j)$. But since $\preceq$ is a wqo, in every infinite sequence $\exists i, j . i < j \wedge p_i \preceq p_j$ (from definition 13(b)), leading to a contradiction. This means a session must be finite, i.e. converge. □

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

Example 4. 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 $exclude(f_1 \cdots f_2 \cdots f_n)$ that exclude programs containing a specific subsequence of function calls (not necessarily consecutive) will be a wqo over the domain of linear programs [13]. In this domain, a user can express feedback such as $exclude(\texttt{close} \cdots \texttt{read})$, thereby ruling out every program that creates a read-after-close error.

5.2 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 14 (Base set). *Let $S \subseteq \mathcal{P}$ be a set of predicates. We define the* base *of S, $Base(S) = \{p \in S \mid \forall p' . p' \Rightarrow_M p \Rightarrow p = p'\}$, 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 15 (Locally strongest user). *Given a candidate program $q_i \notin M^*$, a* locally strongest user *will answer with A_i such that $A_i \cap \bigcup_{m' \in M^*} Base(diff(q_i, m')) \neq \emptyset$. That is, at least one predicate in the answer A_i will be taken from $Base(diff(q_i, m'))$ of some target program m' (where the latter means that no stronger predicate exists in $diff(q_i, m')$).*

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 [26], given a choice between two sequence exclusion predicates `exclude(`$drop$`)` and `exclude(`$drop \cdot take$`)`, 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 $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m')$ 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 13(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 3. *Let $p \preceq p' \iff p \Rightarrow_M p'$. If $\preceq$ is a wfo over $\bigcup_{m' \in M^*} \beta(m')$, then whenever $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m') \neq \emptyset$, we have that $\bigcup_{m' \in M^*} \mathit{Base}(\mathit{diff}(q_i, m')) \neq \emptyset$ 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 $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m')$ for any $q_i \notin M^*$. This is immediate from the property that $\mathit{diff}(q_i, m') \subseteq \beta(m')$ and hence $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m') \subseteq \bigcup_{m' \in M^*} \beta(m')$. Since $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m')$ is nonempty, well foundedness ensures that its base set is also nonempty, and hence also $\bigcup_{m' \in M^*} \mathit{Base}(\mathit{diff}(q_i, m')) \neq \emptyset$. □

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 2. *If $\bigcup_{m' \in M^*} \beta(m')$ is a wfo, $\bigcup_{m' \in M^*} \mathit{Base}(\beta(m'))$ 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 $\bigcup_{m' \in M^*} \mathit{Base}(\beta(m'))$ is similar to a wqo's finite basis requirement (Higman [13]). 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 1, this would already be true because of the finite basis property.

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

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

- $\gamma(S_n) \subseteq M^*$ (as will be seen later in definition 16, $S_n = B \in \mathcal{B}$), and the session has converged successfully, or
- $\gamma(S_n) = \emptyset$, which means $q_{n+1} = \bot$, or the session has converged unsuccessfully.

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 $\bigcup_{m' \in M^*} Base(\mathit{diff}(q_i, m'))$, and other correct user actions may still lead to a contradiction. Second, throughout the session, the user may select predicates from $Base(\mathit{diff}(q_i, m')) \subseteq \beta(m')$ of a different m', and these predicates may contradict. The latter is no longer a possibility if M^* is a singleton set. □

Example 5. Let us assume a singleton $M^* = \{m^*\}$, a domain of functional programs over a vocabulary $\mathcal{V}$ and a set of syntactic predicates $\mathcal{P} = \{include(seq), exclude(seq)\}$ predicates over all continuous sequences of methods $seq = f_1 \cdot f_2 \cdots f_n \in \mathcal{V}$.

We can see immediately that $\mathcal{P}$ itself is not a wfo: for every sequence used by $include$, 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 $exclude$ sequences can longer than the length of m^* as long as we wish and will still appear in $\beta(m^*)$, $include$ sequences that are longer than m^* will rule out m^*. This means that the chain of $include$ predicates in $\beta(m^*)$ is finite, and so $\beta(m^*)$ has a well-founded ordering.

6 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 3). 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 16 (Core set). *We say that a set $B \subseteq \mathcal{P}$ 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, $\mathcal{B} = \{B \subseteq \mathcal{P} \mid \emptyset \neq \gamma(B) \subseteq M^* \wedge |B| \in \mathbb{N}\}$.*

If there exists no $B \in \mathcal{B}$ such that $\emptyset \neq \gamma(B) \subseteq M^*$, then there is no finite under-approximation of the target space in the abstract domain defined by $\mathcal{P}$. 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 17 ($\mathcal{P}$-realizability). *We say that M^* is* $\mathcal{P}$-realizable *if $\mathcal{B} \neq \emptyset$.*

Indeed, $\mathcal{P}$-realizability is a necessary condition for successful convergence. For example example 3 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), 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 18 (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 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 ⇒ 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 $S_{m-1} \sqsubseteq S_{n-1}$, then $\gamma(S_{m-1}) \subseteq \gamma(S_{n-1})$ (Galois connection).

1 ⇒ 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: $S'_n = S_{n-1} \sqcap A_n = S_{n-1} \sqcap B$, 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 $\mathcal{S}$ is complete), the extension $\mathcal{S}'$ of $\mathcal{S}$ constructed from the non-inevitable failure point by selecting $A_n = B$ constitutes both weak and strong progress. The reason is that for $q_n \notin M^*$, $q_n \nvDash B$, 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 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^*).

6.1 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 `if` expressions, equality (==), all list constants over integers (e.g. `[]`, `[1, 2, 3]` etc.), recursive call f, the input variable `i`, and the library functions `cons`, `max`, `remove`, `sort`, and `reverse`.

The predicate set $\mathcal{P}$ contains all input-output examples (ι, ω), and syntactic exclusion of a single element, that is "exclude e" for $e \in \{$`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.

i	A_{i-1}	q_i
1	$([], [])$ $([1,2], [2,1])$	`reverse(i)`

2	exclude `reverse`	**`if`** `(i == [1, 2]) [2, 1]`
		`else` `i`

3	$([1,3], [3,1])$	**`if`** `(i == [1, 2]) [2, 1]`
		`else if` `(i == [1, 3]) [3, 1]`
		`else` `i`

⋮		
n	exclude ==	$\bot$

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 `reverse`. The synthesizer then, quite unfortunately, takes the path of over-fitting the example set via branching using the `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 $Select$ returns $\bot$.

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

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

To see why this is the core set, first note that the exclusion of either `if` or ==, 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 `remove` to reorder the elements. [3] Moreover, when excluding neither `if` nor ==, 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 `reverse`, since any 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 ($\notin M$).

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 $\vDash A_{2..(n-2)}$. 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 4. *For any given $k \in \mathbb{N}$, there exist:*

1. *a session $\mathcal{S}$ of length $k + i$ where S_i is an inevitable failure point and $q_{k+i} = \bot$.*
2. *a session $\mathcal{S}$ where S_i is an inevitable failure point and S_{k+i} is the first infeasible point.*

[3] The number of `removes` has to be at least $\left|\langle x \in \iota_1 \mid x > \text{head}(\iota_1)\rangle\right|$, but at most $|\iota_2|$, which is not possible without branches.

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 nth iteration exhibits both a first infeasible point and failure with $\bot$.

7 Discussion

In this section we discuss the implication of some of the conditions posed in definitions and theorems in the previous sections.

7.1 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 [20] and BlinkFill [30] follow the same workflow. GIM [26] 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 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 [24], 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 [26] 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 1, 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, 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.

7.2 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 6.1. 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 [22] and for heuristically arriving at the user's intentions faster [20]. All that remains for the synthesizer to do is to better communicate the limitations of M.

7.3 Sharing more with the user

One of the design tenets behind [26] 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. Section 7.1 suggesting 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.

8 Related Work

Programming by Example In PBE the interaction between user and synthesizer for demonstrating the desired behavior is restricted to examples, both in initial specifications and any refinement. FlashFill [10, 29] is a PBE tool for automating transformations on an Excel data set, and is included in Microsoft Excel. Its implementation is based on the theory of Version-Space Algebra [19]. FlashFill is iterative by design, accepting a (strong progress) update to its specification if the resulting program is not satisfactory. The FlashMeta family of synthesizers [20, 29, 30] follow this same trend.

Counterexample-guided inductive synthesis CEGIS is a synthesis framework that has been formalized in [31] and [21]. It is implemented in tools such as Sketch [32, 33], 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 [1] 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 In type-directed synthesis tools such as [9, 12, 27], the specification is provided entirely by types. These tools tend to not use an iterative model, as refining the specification is not trivial. Synquid [28] 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 [6, 7, 24] 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 [1], VSA-based synthesizers [29], and oracle-driven synthesizers via inductive learning [14]. These all describe a single-iteration interaction with the user (though [14], 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 [21] 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 [29]. The work of Loding et al. [22] 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 this paper. 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.

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.

Bibliography

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.

[2] T. Anton. Xpath-wrapper induction by generalizing tree traversal patterns. In *Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken*, pages 126–133, 2005.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[4] D. Drachsler-Cohen, S. Shoham, and E. Yahav. Synthesis with abstract examples. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 254–278, 07 2017.

[5] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[6] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2017*, 2017.

[7] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[9] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

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

[12] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[13] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

[14] S. Jha and S. A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, Feb 2017.

[15] D. E. Knuth. Computer programming as an art. In *ACM Turing award lectures*, page 1974. ACM, 2007.

[16] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.

[17] J. Landauer and M. Hirakawa. Visual awk: A model for text processing by demonstration. In *vl*, pages 267–274, 1995.

[18] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Learning repetitive text-editing procedures with smartedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, pages 209–226, 2001.

[19] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1):111–156, 2003.

[20] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

[21] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani. Interactive program synthesis, 2017.

[22] C. Löding, P. Madhusudan, and D. Neider. Abstract learning frameworks for synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–185. Springer, 2016.

[23] A. Omari, S. Shoham, and E. Yahav. 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*, pages 368–379, 2016.

[24] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.

[25] H. Peleg, S. Shoham, and E. Yahav. D3: Data-driven disjunctive abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 185–205. Springer, 2016.

[26] H. Peleg, S. Shoham, and E. Yahav. Programming not only by example, 2017.

[27] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Notices*, volume 47, pages 275–286. ACM, 2012.

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

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

[30] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. September 2016.

[31] A. Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[32] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, volume 43, pages 136–148. ACM, 2008.

[33] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

[34] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.

[35] I. H. Witten and D. Mo. Tels: Learning text editing tasks from examples. In *Watch what I do*, pages 183–203. MIT Press, 1993.

[36] S. Wu, J. Liu, and J. Fan. Automatic web content extraction by combination of learning and grouping. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1264–1274. ACM, 2015.