Bottom-Up Synthesis of Recursive Functional Programs using Angelic Execution

ANDERS MILTNER, UT Austin, USA
ADRIAN TREJO NUÑEZ, UT Austin, USA
ANA BRENDEL, UT Austin, USA
SWARAT CHAUDHURI, UT Austin, USA
ISIL DILLIG, UT Austin, USA

We present a novel bottom-up method for the synthesis of functional recursive programs. While bottom-up synthesis techniques can work better than top-down methods in certain settings, there is no prior technique for synthesizing recursive programs from logical specifications in a purely bottom-up fashion. The main challenge is that effective bottom-up methods need to execute sub-expressions of the code being synthesized, but it is impossible to execute a recursive subexpression of a program that has not been fully constructed yet. In this paper, we address this challenge using the concept of *angelic semantics*. Specifically, our method finds a program that satisfies the specification under angelic semantics (we refer to this as *angelic synthesis*), analyzes the assumptions made during its angelic execution, uses this analysis to strengthen the specification, and finally reattempts synthesis with the strengthened specification. Our proposed angelic synthesis algorithm is based on version space learning and therefore deals effectively with many incremental synthesis calls made during the overall algorithm. We have implemented this approach in a prototype called Burst and evaluate it on synthesis problems from prior work. Our experiments show that Burst is able to synthesize a solution to 94% of the benchmarks in our benchmark suite, outperforming prior work.

CCS Concepts: • **Software and its engineering → Recursion**; *Functional languages*; • **Theory of computation → Tree languages**.

Additional Key Words and Phrases: Program Synthesis, Angelic Execution, Logical Specifications

ACM Reference Format:
Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-Up Synthesis of Recursive Functional Programs using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (January 2022), 29 pages. https://doi.org/10.1145/3498682

1 INTRODUCTION

Methods for program synthesis from formal specifications typically come in two flavors: *top-down* and *bottom-up*. Top-down methods [Feser et al. 2015; Frankle et al. 2016; Gulwani 2011; Kitzelmann et al. 2006; Osera and Zdancewic 2015; Polikarpova et al. 2016; Summers 1977] iterate through a sequence of *partial programs*, starting with an "empty" program and progressively refining them through the addition of new code. In contrast, bottom-up methods [Albarghouthi et al. 2013; Alur et al. 2015; Odena et al. 2020; Udupa et al. 2013] maintain a pool of *complete programs* and progressively generate new programs by composing existing ones.

Authors' addresses: Anders Miltner, UT Austin, Austin, TX, USA, amiltner@cs.utexas.edu; Adrian Trejo Nuñez, UT Austin, Austin, TX, USA, atrejo@cs.utexas.edu; Ana Brendel, UT Austin, Austin, TX, USA, anabrendel@utexas.edu; Swarat Chaudhuri, UT Austin, Austin, TX, USA, swarat@cs.utexas.edu; Isil Dillig, UT Austin, Austin, TX, USA, isil@cs.utexas.edu.

2475-1421/2022/1-ART21
https://doi.org/10.1145/3498682

Top-down and bottom-up approaches have complementary strengths. For example, top-down methods work well when the specification can be naturally decomposed into subgoals through an analysis of partial programs. However, they can run into imprecision or computational complexity issues when the specification or the language semantics are complicated. In contrast, a bottom-up approach only needs to evaluate complete sub-expressions of a program, which is generally a much easier task than that of reasoning about partial programs.

Unfortunately, it is difficult to apply bottom-up synthesis to programming languages that permit recursion. This is because effective bottom-up approaches need to execute all sub-expressions of the target program; however, for recursive programs, sub-expressions can call the function being synthesized, whose semantics are still unknown. One way to overcome this issue is to assume that the specification is *trace-complete*, i.e., that the result of each such evaluation is part of the specification. (Indeed, such a strategy is followed in the Escher [Albarghouthi et al. 2013] system for the bottom-up synthesis of recursive programs.) However, trace-completeness is a restrictive assumption, and writing trace-complete specifications can be cumbersome and unintuitive.

In this paper, we propose a new approach to bottom-up program synthesis that addresses this difficulty. The key insight behind our solution is to use *angelic execution* [Broy and Wirsing 1981] to evaluate recursive sub-expressions of the program being synthesized. Specifically, our method first performs *angelic synthesis* to find a program P that satisfies the specification under the assumption that recursive calls can return *any* value that is consistent with the specification. For example, if the specification is $0 \leq f(x) \leq x$, the angelic synthesizer assumes that a recursive call $f(2)$ can return *any* of the integers 0, 1, or 2, although in reality it can only return *one* of these. Thus, when performing angelic synthesis of a function f, we only need access to f's *specification* rather than its full *implementation*.

One complication with this approach is that a program P that angelically satisfies its specification φ may not *actually* satisfy φ. To deal with this difficulty, our method combines angelic synthesis with *specification strengthening* and back-tracking search. In more detail, given an angelic synthesis result P, our synthesis technique first checks if P satisfies φ under the standard semantics. If so, then P is returned as a solution. Otherwise, our method analyzes the assumptions made in angelic executions of P, uses this information to strengthen the specification, and re-attempts synthesis with the strengthened specification. If synthesis is unsuccessful with the strengthened specification, it backtracks and tries a different strengthening, continuing this process until it either finds the right program or exhausts the search space.

As illustrated by the above discussion, our end-to-end approach requires gradually strengthening the specification and making many calls to an *angelic synthesizer*. Thus, for our approach to be practical, it is important to have an angelic synthesis technique that can reuse partial synthesis results. Additionally, it must be possible to easily analyze assumptions made in angelic executions in order to determine how to strengthen the specification. Motivated by these considerations, we propose an angelic synthesis technique based on *finite tree automata* [Wang et al. 2017a,b]. Our proposed angelic synthesizer handles incremental specifications by taking the intersection of previously constructed tree automata (for weaker specifications) with new automata constructed from the additional specifications. This incremental nature of the angelic synthesizer allows our approach to efficiently handle a series of increasingly more complex specifications. Furthermore, by inspecting runs of the tree automaton, we can easily and efficiently analyze the assumptions made by the angelic synthesizer.

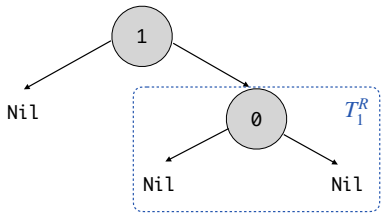

Fig. 1. First example T_1

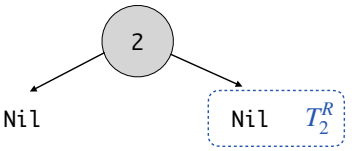

Fig. 2. Second example T_2

We have implemented our technique in a tool called Burst[1] and evaluate it on 45 benchmarks from prior work [Osera and Zdancewic 2015] using three types of specifications, namely (1) input-output examples, (2) reference implementations, and (3) logical formulas. Our evaluation shows that Burst can synthesize more functions than prior work on all three types of specifications. In particular, our tool is able to synthesize 96% (43) of the functions from input/output examples, 96% (43) of the functions from reference implementations, and 91% (41) of the functions from logical specifications. We also compare Burst against a simpler variant that does not perform specification strengthening, and we show that our proposed backtracking search technique is useful in practice.

In summary, this paper makes the following contributions:

- We present the first bottom-up synthesis procedure that can handle general recursion and general logical specifications, and does not require the restrictive trace-completeness assumption.
- We introduce a new form of *angelic program synthesis* that combines the use of angelic program semantics and specification strengthening and can make use of efficient version space representations. Some of the insights in our algorithm may be applicable outside the immediate setting that we target.
- Our artifactual contribution is an implementation of our approach, called Burst. We have conducted an extensive experimental evaluation on synthesis benchmarks from prior work. Our experiments show that Burst significantly outperforms the state-of-the-art in the synthesis of recursive programs on several counts.

2 OVERVIEW

In this section, we give an overview of our method with the aid of a motivating example. Our goal in this example is to synthesize a recursive implementation of the `right_spine` procedure, which takes as input a tree and produces a list that is obtained by traversing the rightmost children of a node, starting from the root and continuing until a leaf node is reached. As an example, Figures 1 and 2 show two trees T_1 and T_2, and a partial input-output specification for `right_spine` is given as follows:

$$\begin{aligned} \texttt{right\_spine}(T_1) &= [1;0] \\ \texttt{right\_spine}(T_2) &= [2] \end{aligned} \tag{1}$$

Note that our method can work with specifications that are not input-output examples (see Section 2.4); here, we simply choose it for simplicity of presentation. We now explain how our technique synthesizes this `right_spine` procedure in a bottom-up fashion.

[1]Bottom-Up Recursive SynThesizer

2.1 High Level Algorithm

Our algorithm works in a refinement loop that performs two major steps: (1) it synthesizes a program that *angelically* satisfies the specification, and (2) strengthens the specification based on the assumptions made in the angelic execution. In this subsection, we illustrate the high-level approach on `right_spine`, leaving the details of angelic synthesis to Section 2.2.

Iteration 1. The algorithm starts by invoking the angelic synthesizer to find a program that *angelically* satisfies the specification shown in Equation 1. As we will discuss later, the angelic synthesizer outputs the following program in this iteration:

```
let rec P1(x) =
        match x with
          | Nil -> []
          | Node(l,v,r) -> P1(r)
```

Clearly, this program does not actually satisfy the specification, but it does satisfy the specification under the angelic semantics: Since the specification from Eq. 1 does not constrain the output of the recursive call on T_1^R, the angelic synthesizer assumes that the recursive call to `P1` can return anything, including `[1;0]`, for the right subtree of T_1. Thus, program `P1` satisfies the specification under the angelic semantics of recursion.

Next, our algorithm checks whether the candidate program satisfies the specification under the actual semantics. Since $\texttt{P1}(T_1) = \texttt{P1}(T_2) = \texttt{[]}$, it clearly does not, and our algorithm analyzes the assumptions made in the angelic execution to determine how to strengthen the specification. In this case, the angelic execution assumes that the recursive call on the right-subtrees T_1^R and T_2^R return $[1;0]$ and $[2]$ respectively. Thus, our algorithm re-attempts synthesis using the following strengthened specification:

$$\begin{aligned} \texttt{right\_spine}(T_1) &= [1;0] \qquad \texttt{right\_spine}(T_2) &= [2] \\ \texttt{right\_spine}(T_1^R) &= [1;0] \qquad \texttt{right\_spine}(T_2^R) &= [2] \end{aligned} \tag{2}$$

Iteration 2a. In the next recursive call, our algorithm invokes the angelic synthesizer to find a program consistent with the specification shown in Eq. 2 but it fails.

Iteration 2b. Since synthesis was unsuccessful for Eq. 2, our algorithm backtracks and tries a different strengthening. Specifically, since we could not find a program where the recursive calls on T_1^R and T_2^R return $[1;0]$ and $[2]$ respectively, we now strengthen the specification using the *negation* of these assumptions. This yields the following specification for the next recursive call to the synthesizer:

$$\begin{aligned} \texttt{right\_spine}(T_1) = [1;0] \qquad & \texttt{right\_spine}(T_2) = [2] \\ \neg(\texttt{right\_spine}(T_1^R) = [1;0] \quad \wedge \quad & \texttt{right\_spine}(T_2^R) = [2]) \end{aligned} \tag{3}$$

In this case, the angelic synthesizer returns the following program:

```
let rec P2(x) =
        match x with
          | Nil -> [0]
          | Node(l,v,r) -> v::P2(r)
```

This program is again incorrect but it does satisfy Eq. 3 under the angelic semantics. Indeed, a "witness" to angelic satisfaction is:

$$\texttt{P2}(T_1^R) = \texttt{[0]} \wedge \texttt{P2}(T_2^R) = \texttt{[]}$$

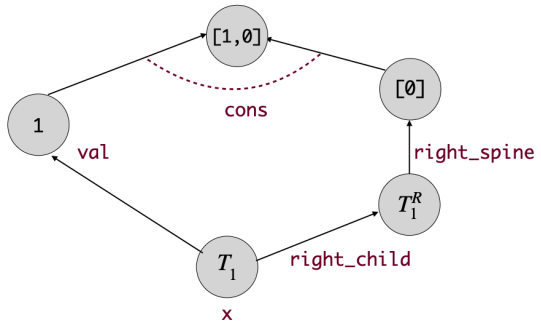

Fig. 3. An example FTA that accepts a program that brings the input T_1 to the valid output of `[1;0]`.

Note that this assumption is allowed under the angelic semantics since these return values on T_1^R and T_2^R are both consistent with Eq. 3. Thus, using the witness to angelic satisfaction, we now strengthen the specification as follows:

$$\begin{array}{rcl} \texttt{right\_spine}(T_1) = [1;0] & & \texttt{right\_spine}(T_2) = [2] \\ \neg(\texttt{right\_spine}(T_1^R) = [1;0] & \wedge & \texttt{right\_spine}(T_2^R) = [2]) \\ \texttt{right\_spine}(T_1^R) = [0] & & \texttt{right\_spine}(T_2^R) = [] \end{array} \tag{4}$$

Iteration 3b. In the next (and last) iteration, when we invoke the angelic synthesizer on Eq. 4, it outputs the following program:

```
let rec P3(x) =
          match x with
            | Nil -> []
            | Node(l,v,r) -> v::P3(r)
```

This program satisfies the specification under the actual semantics; thus, the algorithm terminates with `P3` as the (correct) solution.

2.2 Angelic Synthesis using FTAs

As illustrated by the above discussion, a key piece of our technique is the *angelic synthesizer* for finding a program that satisfies the specification under the angelic semantics. Inspired by prior work on bottom-up synthesis [Wang et al. 2017a,b], our angelic synthesizer constructs a *finite tree automaton (FTA)* that compactly represents a set of programs. In a nutshell, FTAs generalize standard automata by accepting trees instead of words. In our setting, the states in the automata correspond to concrete program values (e.g., lists like `[1;0]` or `[]`), and the trees accepted by the automaton correspond to programs (i.e., abstract syntax trees).

In order to explain our angelic synthesis approach, we first briefly review the construction from prior work [Wang et al. 2017b]. The idea is to construct a separate automaton for each input (e.g., T_1 from Fig 1) and then take the intersection of all of these automata. To construct each automaton, we start with the given input and obtain new states by applying language constructs to the existing states. For example, given states q_2 and q_3 representing integers 2 and 3 and the operator +, we generate a new state q_5 (for integer 5) by applying the transition $+(q_2, q_3) \rightarrow q_5$. Since the accepting states of the FTA are those that satisfy the specification, the language of the constructed automaton includes exactly those programs that are consistent with the specification.

As illustrated by the above discussion, such an FTA-based synthesis method is *bottom-up* in that it evaluates complete sub-expressions on the input and combines the values of these sub-expressions to generate new values. However, prior work cannot deal with recursive functions because it is not possible to evaluate a function that has not yet been synthesized. For example, consider the recursive call to `right_spine(r)` where r has value T_1^R in our running example. Since `right_spine` has not yet been synthesized, we simply do not know what `right_spine` will return on T_1^R.

To deal with this challenge, our angelic synthesizer assumes that the result of the recursive call could be any value that is consistent with the specification. In particular, given a specification φ and FTA states $q_1, \ldots, q_n$, we assume that a recursive invocation expression $\texttt{f}(\overline{q})$ could evaluate to any q_i as long as q_i is consistent with φ. For instance, Figure 3 shows an FTA with states $\{T_1, T_1^R, 1, [0], [1;0]\}$ for the angelic synthesis problem for Eq. 3. Here, there is a transition $\texttt{right\_spine}(T_1^R) \rightarrow [0]$ since the call $\texttt{right\_spine}(T_1^R) = [0]$ is consistent with Eq. 3. Note that edges in Figure 3 correspond to program syntax and `[1,0]` is an accepting state, so the program `right_spine(x) = val(x)::right_spine(right_child(x)))` is accepted by this FTA.

As illustrated by this discussion, the use of angelic semantics allows us to construct a bottom-up tree automaton despite not knowing what the recursive invocation will return on a given input. However, an obvious ramification of this is that programs accepted by the automaton may not satisfy the specification under the true semantics, which is why our method combines angelic synthesis with specification strengthening and backtracking search, as described in Section 2.1.

2.3 Incremental Synthesis

As we saw from Equations 1, 3, and 4 from Section 2.1, successive calls to the synthesis algorithm involve increasingly strong specifications. In particular, if the synthesis algorithm is invoked on specification φ in the i'th iteration, then the specification in the $i+1$'th iteration is of the form $\varphi \wedge \psi$. We exploit this incremental nature of the algorithm to make angelic synthesis more efficient.

In particular, recall that our angelic synthesizer based on FTAs constructs a different FTA for each input and then takes their intersection. Thus, given a specification $\varphi \wedge \psi$ where φ is the old specification, we can simply construct a new FTA for ψ and then take its intersection with the old FTA for φ. Hence, performing angelic synthesis using FTAs allows us to reuse all the work from prior iterations.

2.4 Generalization to Arbitrary Logical Specs

In our example so far, we illustrated the synthesis algorithm on the simple input-output examples from Eq. 1. However, our method can be generalized to more complicated logical specifications using the standard counterexample-guided inductive synthesis (CEGIS) paradigm. In particular, since our core synthesis algorithm takes as input *ground formulas* (defined in Section 3) as opposed to input-output examples, it can be easily incorporated within the CEGIS loop to handle more general logical specifications. For instance, our method can produce the correct implementation of `right_spine` given the following logical specification:

$$\phi(in, out) := \texttt{no\_left\_subchildren}(in) \Rightarrow (\texttt{tree\_size}(in) == \texttt{list\_size}(out))$$

where `tree_size` and `list_size` return the number of elements in a tree and list respectively, and `no_left_subchildren` returns true if the left child of every node in the tree is a leaf.

3 PROBLEM STATEMENT

In this section, we present our problem statement, which is synthesizing recursive programs in a simple ML-like language with products and sums (see Figure 4). Without loss of generality, we

$$
\begin{array}{lll}
P & ::= & \texttt{rec f}(x) = e \\
e & ::= & x \\
 & | & e_1\ e_2 \qquad\qquad | \ \texttt{unit} \\
 & | & \texttt{inl}\ e \qquad\qquad | \ \texttt{inr}\ e \\
 & | & \texttt{unl}\ e \qquad\qquad | \ \texttt{unr}\ e \\
 & | & \texttt{fst}\ e \qquad\qquad | \ \texttt{snd}\ e \\
 & | & (e_1,\ e_2) \qquad | \ \texttt{switch}\ e_3\ \texttt{on inl}\ \_ \to e_1 \mid \texttt{inr}\ \_ \to e_2 \\
v & ::= & \texttt{unit} \qquad | \ (v_1,\ v_2) \\
 & | & \texttt{inl}\ v \qquad | \ \texttt{inr}\ v
\end{array}
$$

Fig. 4. A functional ML-like language with explicit recursion in which we synthesize programs. The nonterminal P denotes programs in this language, and the nonterminal v denotes values in this language.

$$
\frac{e_2 \Downarrow v_2 \qquad e_1[\texttt{rec f}(x) = e_1/\texttt{f}, v_2/x] \Downarrow v_3}{(\texttt{rec f}(x) = e_1)\ e_2 \Downarrow v_3}
$$

$$
\frac{}{\texttt{unit} \Downarrow \texttt{unit}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{snd}\ e \Downarrow v_2}
$$

$$
\frac{e \Downarrow v}{\texttt{inl}\ e \Downarrow \texttt{inl}\ v} \qquad \frac{e \Downarrow v}{\texttt{inr}\ e \Downarrow \texttt{inr}\ v} \qquad \frac{e \Downarrow \texttt{inl}\ v}{\texttt{unl}\ e \Downarrow v} \qquad \frac{e \Downarrow \texttt{inr}\ v}{\texttt{unr}\ e \Downarrow v}
$$

$$
\frac{e_3 \Downarrow \texttt{inl}\ v_3 \qquad e_1 \Downarrow v_1}{\texttt{switch}\ e_3\ \texttt{on inl}\ \_ \to e_1\ \texttt{inr}\ \_ \to e_2 \Downarrow v_1} \qquad \frac{e_3 \Downarrow \texttt{inr}\ v_3 \qquad e_2 \Downarrow v_2}{\texttt{switch}\ e_3\ \texttt{on inl}\ \_ \to e_1\ \texttt{inr}\ \_ \to e_2 \Downarrow v_2}
$$

Fig. 5. Program Semantics. The symbols e range over expressions and v range over values. Both f and x denote arbitrary free variables. If $P = \texttt{rec f}(x) = e$ and $P\ v \Downarrow v'$ then $[[P]](v) = v'$.

assume that programs take a single input, as we can represent multiple inputs using tuples (i.e., pairs with nested pairs). Given a program P and a concrete input v, we use the notation $[[P]](v)$ to denote the result of executing P on input v according to the semantics presented in Figure 5.

Our goal in this paper is to synthesize a *single* recursive procedure f from a given specification, which is represented as a *ground formula* φ. We assume that φ always contains a special uninterpreted function symbol f which refers to the function to be synthesized. More formally, we define *ground specifications* as follows:

Definition 3.1. **(Ground specification)** A *ground specification* is a boolean combination of atomic formulas of the form $f(i)$ op c where f denotes the function to be synthesized, i and c are constants (with i being the input), and op is a binary relation.

Definition 3.2. **(Satisfaction of ground spec)** Given a program P defining function f, we say that P *satisfies* a ground specification φ, denoted $P \models \varphi$, iff the following condition holds:

$$
P \models \varphi \iff \models \varphi\big[[[P]](x)/f(x)\big]
$$

where the notation $\varphi[[[P]](x)/f(x)]$ denotes the formula φ with every ground term $f(v_i)$ is replaced by $[[P]](v_i)$.

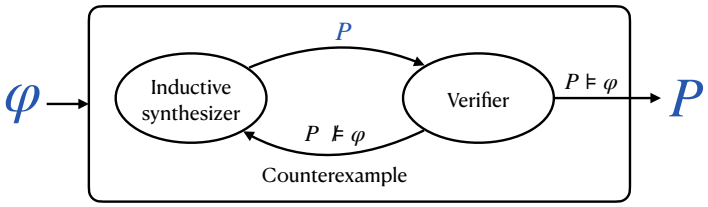

Fig. 6. Counterexample-guided inductive synthesis. Since the input to the inductive synthesizer is a ground formula, our approach can be lifted to a general class of specifications using the CEGIS paradigm.

Problem Statement: Given a ground specification φ, find program P such that $P \models \varphi$.

Note that ground specifications are quite powerful: synthesizers that can generate programs from ground specifications can also perform synthesis from a number of specification classes. For example we can always encode I/O examples as ground formulas. but not vice versa.

Example 3.3. Consider the set of input-output examples $\{1 \mapsto 2, 2 \mapsto 3\}$. We can encode this specification in our format using the ground formula $f(1) = 2 \wedge f(2) = 3$. In general, I/O examples correspond to specifications of the form:

$$\bigwedge_k f(i_k) = o_k$$

where (i_k, o_k) are input-output pairs.

Furthermore, we can also lift synthesis from ground specifications to an even more general class of specifications using the well-known CEGIS paradigm (see Figure 6). Given a fixed set of inputs I and a general predicate φ with variables (representing inputs), one can convert this into a ground formula of the form $\bigwedge_{i \in I} \varphi(i)$ where each i a counterexample returned by the verifier. Since the CEGIS paradigm invokes the verifier to add new counterexamples if the synthesized program is not correct, synthesis from ground formulas immediately provides a way to perform synthesis from more general logical specifications.

Example 3.4. Consider the problem of synthesizing a function that returns a value greater than its input for all positive inputs. The specification for such a function is of the form $x > 0 \Rightarrow f(x) > x$. While this specification is not a ground formula, we can embed our synthesis technique into the CEGIS paradigm and reduce it to inductively synthesizing programs from ground specifications of the form:

$$\bigwedge_k f(i_k) > i_k$$

where each i_j is a positive integer returned as a counterexample by a verifier.

4 ANGELIC RECURSION

As mentioned earlier, our method is based on bottom-up synthesis, which requires the ability to execute sub-expressions of the program being synthesized. Since this is not feasible for recursive procedures, we introduce the notion of *angelic recursion* and *angelic satisfaction*.

Definition 4.1. **(Angelic recursion)** Given a recursive procedure P, the *angelic semantics* of P with respect to specification φ, denoted $[[P]]^{\varphi}$, is defined in Figure 7. These semantics are very similar to the semantics in Figure 5; the key difference lies in how recursion is performed. When

$$\frac{e \Downarrow^{\varphi} v \qquad \mathrm{SAT}(f(v) = v' \wedge \varphi)}{\texttt{f}\ e \Downarrow^{\varphi} v'} \qquad \frac{}{\texttt{unit} \Downarrow^{\varphi} \texttt{unit}} \qquad \frac{e_1 \Downarrow^{\varphi} v_1 \qquad e_2 \Downarrow^{\varphi} v_2}{(e_1, e_2) \Downarrow^{\varphi} (v_1, v_2)}$$

$$\frac{e \Downarrow^{\varphi} (v_1, v_2)}{\texttt{fst}\ e \Downarrow^{\varphi} v_1} \qquad \frac{e \Downarrow^{\varphi} (v_1, v_2)}{\texttt{snd}\ e \Downarrow^{\varphi} v_2}$$

$$\frac{e \Downarrow^{\varphi} v}{\texttt{inl}\ e \Downarrow^{\varphi} \texttt{inl}\ v} \qquad \frac{e \Downarrow^{\varphi} v}{\texttt{inr}\ e \Downarrow^{\varphi} \texttt{inr}\ v} \qquad \frac{e \Downarrow^{\varphi} \texttt{inl}\ v}{\texttt{unl}\ e \Downarrow^{\varphi} v} \qquad \frac{e \Downarrow^{\varphi} \texttt{inr}\ v}{\texttt{unr}\ e \Downarrow^{\varphi} v}$$

$$\frac{e_3 \Downarrow^{\varphi} \texttt{inl}\ v_3 \qquad e_1 \Downarrow^{\varphi} v_1}{\texttt{switch}\ e_3\ \texttt{on inl}\ x_1 \rightarrow e_1\ \texttt{inr}\ x_2 \rightarrow e_2 \Downarrow^{\varphi} v_1}$$

$$\frac{e_3 \Downarrow^{\varphi} \texttt{inr}\ v_3 \qquad e_2 \Downarrow^{\varphi} v_2}{\texttt{switch}\ e_3\ \texttt{on inl}\ x_1 \rightarrow e_1\ \texttt{inr}\ x_2 \rightarrow e_2 \Downarrow^{\varphi} v_2}$$

$$\frac{P = \texttt{rec f}(x) = e \qquad e[v/x] \Downarrow^{\varphi} v'}{v' \in [[P]]^{\varphi}(v)}$$

Fig. 7. Angelic Semantics. The key difference between angelic semantics and standard semantics lies in the first rule, for recursive calls.

performing a recursive call $\texttt{f}(v)$, the result can be any v' where $f(v) = v'$ is consistent with the specification. Thus, $[[P]]^{\varphi}(v)$ yields a *set* of values $\mathcal{V}$.

Intuitively, angelic recursion is useful in our setting because it allows us to "execute" a recursive program without knowing the exact behavior of recursive calls.

Example 4.2. Let P be the program `rec f(x) = if x=0 then 1 else f(x-1)` and suppose that $\varphi = f(0) > 0$. Then, $[[P]]^{\varphi}(0) = \{1\}$, and $[[P]]^{\varphi}(1) = \{y \mid y > 0\}$. In particular, for input 1, f contains a recursive invocation on input 0, and the angelic semantics allows the recursive call to return any value greater than 0. Thus, $[[P]]^{\varphi}(1)$ is exactly the set of positive integers.

Next, we define a notion of angelic satisfaction:

Definition 4.3. **(Angelic satisfaction on input)** Given a program P defining function f, we say that P *angelically satisfies* specification φ on input v, denoted $P \models^{😇}_{v} \varphi$, iff the following condition holds:

$$P \models^{😇}_{v} \varphi \iff \exists v'.\ v' \in [[P]]^{\varphi}(v) \wedge \mathrm{SAT}(f(v) = v' \wedge \varphi)$$

Next, we generalize this notion of angelic satisfaction from a single input to all inputs:

Definition 4.4. **(Angelic satisfaction)** A program P *angelically satisfies* specification φ, denoted $P \models^{😇} \varphi$, iff for all possible inputs v, we have $P \models^{😇}_{v} \varphi$.

Note that angelic satisfaction ($P \models^{😇} \varphi$) is a much weaker notion than standard satisfaction ($P \models \varphi$). This is illustrated by the following example:

Example 4.5. Let P be the program `rec f(x) = if x=0 then 1 else f(x-1)` and suppose that $\varphi = f(0) > 0 \wedge f(1) > 1$. Then, $P \models^{😇} \varphi$, as $\forall x.\ x + 1 \in [[P]]^{\varphi}(x)$. However, clearly, this program does not satisfy φ with respect to the standard semantics because we have $[[P]](1) = 1$.

input: Ground specification χ
output: A program P or $\bot$
global: Ω is a learned *anti-specification*, initially $\emptyset$

```
1:  procedure Synthesize(χ)
2:      result ← SynthesizeAngelic(χ ∧ ⋀_{φ_i ∈ Ω} ¬φ_i)
3:      match result with
4:          | Failure(κ) →
5:              Ω ← Ω ∪ κ
6:              return ⊥
7:          | Success(P, ω) →
8:              if P ⊨ χ then return P
9:              else
10:                 P ← Synthesize(χ ∧ ω)
11:                 if P = ⊥ then return Synthesize(χ ∧ ¬ω)
12:                 else return P
```

Algorithm 1. Core Recursive Synthesis Algorithm

If a program P angelically satisfies a specification φ, we can define a *witness* to angelic satisfaction as follows:

Definition 4.6. **(Witness to angelic satisfaction)** Let P be a program such that $P \models^{😇} \varphi$. Then, a *witness* ω to angelic satisfaction of P is a formula $\bigwedge_i f(c_i) = c_i'$ such that, if $P \models \omega$, then $P \models \varphi$.

Intuitively, a *witness* to angelic satisfaction specifies what the recursive calls in P must return in order for P to actually satisfy the specification. We discuss how to find these witnesses in Section 6.2.2.

Example 4.7. Consider the program `rec f(x) = if x=0 then 1 else f(x-1)+1` and the ground specification $f(1) > 1 \wedge f(2) > 2$. This program angelically satisfies the specification in an execution where the recursive call returns $f(0) = 1$ and $f(1) = 2$. Thus, $f(0) = 1 \wedge f(1) = 2$ is a witness to angelic satisfaction. Of course, note that angelic witnesses are not unique. For example, $f(0) = 2 \wedge f(1) = 3$ is also a witness to angelic satisfaction.

5 SYNTHESIS ALGORITHM USING ANGELIC EXECUTION

In this section, we describe our top-level synthesis algorithm based on angelic recursion. While this synthesis algorithm does not specify whether to construct programs in a top-down or bottom-up fashion, we emphasize that it is the use of *angelic recursion* that makes it possible to implement its key components using a bottom-up approach (as we discuss in the next section).

Algorithm 1 shows the high-level structure of our synthesis algorithm. The procedure Synthesize takes as input a ground specification χ and returns either a program P or $\bot$ to indicate that synthesis is unsuccessful. Internally, the algorithm also maintains a global variable, namely set Ω, that we refer to as an *anti-specification* which is used for pruning the search space. In particular, Ω is constructed in such a way that any program that satisfies $\phi \in \Omega$ is guaranteed to *not* satisfy the desired specification χ, i.e.:

$$\forall P.\ \forall \phi \in \Omega.\ P \models \phi \Rightarrow P \not\models \chi \tag{5}$$

Since the contrapositive of Equation 5 is

$$\forall P.\ \forall \phi \in \Omega.\ P \models \chi \Rightarrow P \not\models \phi$$

which implies that P must satisfy $\bigwedge_{\phi_i \in \Omega} \neg\phi_i$ in order to also satisfy χ, we can use Ω to construct a stronger specification and thereby reduce the search space.

Our synthesis procedure starts by invoking a procedure call SynthesizeAngelic (line 2) which takes as input a specification that the returned program must satisfy under the angelic semantics. In particular, given a (ground) specification φ, SynthesizeAngelic either returns failure or a program P that *angelically* satisfies φ (i.e., $P \models^{😇} \varphi$). If the output is failure (meaning that there is no program in the search space that satisfies φ), SynthesizeAngelic also returns an anti-specification (i.e., set of formulas) κ that serves as an "explanation" of why angelic synthesis failed. In particular, κ has the property, for every $\psi \in \kappa$, there is no program in the search space that satisfies ψ. Thus, if SynthesizeAngelic returns $\texttt{Failure}(\kappa)$, we add κ to Ω (line 5).

In the extended example shown in Section 2, there was a failure in Iteration 2a. Our underlying synthesizer would identify that this failure was due to the constraints ($\texttt{right\_spine}(T_2) = [2]$ and $\texttt{right\_spine}(T_2^R) = [2]$). Including this anti-specification would yield the following stronger specification for Iteration 2b:

$$\begin{array}{rcl} \texttt{right\_spine}(T_1) = [1;0] & & \texttt{right\_spine}(T_2) = [2] \\ \neg(\texttt{right\_spine}(T_1^R) = [1;0] & \wedge & \texttt{right\_spine}(T_2^R) = [2]) \\ \neg(\texttt{right\_spine}(T_2) = [2] & \wedge & \texttt{right\_spine}(T_2^R) = [2]) \end{array} \tag{6}$$

If SynthesizeAngelic returns a program P, our synthesis procedure checks whether P satisfies the specification χ under the true semantics (line 8). If so, then it returns P as a valid solution to the synthesis problem. Otherwise, it uses the witness ω to angelic satisfaction returned by SynthesizeAngelic to construct a stronger specification. In particular, recall that such a witness ω encodes assumptions that an angelic execution makes in order to satisfy the specification. Thus, we strengthen the specification as $\chi \wedge \omega$ and re-attempt synthesis by recursively invoking Synthesize on this stronger specification (line 10). If synthesis is successful, we return the resulting program as a solution (line 12); otherwise, we backtrack and recursively invoke Synthesize with the alternative specification $\chi \wedge \neg\omega$ (line 11), which ends up ruling out ω from the search space. Observe that the anti-specification Ω also grows during the recursive calls; thus, the second recursive call at line 11 actually prunes more programs than just those satisfying ω.

The following theorems state the soundness and completeness of our synthesis algorithm.

Theorem 5.1. **(Soundness)** *If* Synthesize(χ) *returns a program P, then we have $P \models \chi$.*

Proof. Follows directly from line 8 of Algorithm 1. □

Theorem 5.2. **(Completeness)** *If* Synthesize(χ) *returns $\bot$, then there is no program that satisfies χ under the assumption that (1)* SynthesizeAngelic *is complete, and (2) if* SynthesizeAngelic *returns* $\texttt{Failure}(\kappa)$*, then κ satisfies the assumption from Equation 5.*

Proof. The proof is in the full version of the paper [Miltner et al. 2021]. □

Remark. A simpler alternative to the specification strengthening approach in Algorithm 1 would be to perform enumerative search over the angelic synthesis results as opposed to strengthening the specification. However, as we show empirically in Section 8, this simpler alternative is not as effective. In particular, many programs that angelically satisfy the specification are wrong due to *shared* incorrect assumptions about recursive calls; thus, our proposed algorithm allows ruling out many incorrect programs at the same time.

6 BOTTOM-UP ANGELIC SYNTHESIS USING TREE AUTOMATA

Recall that our top-level synthesis procedure (Algorithm 1) uses a key procedure called SynthesizeAngelic to find a program that satisfies the specification under the angelic semantics. In this section, we describe a realization of the angelic synthesis algorithm using bottom-up finite tree automata. Towards this goal, we first review tree automata basics and then describe the angelic synthesis algorithm.

6.1 Tree Automata Preliminaries

A *finite tree automaton* is a state machine that describes sets of trees [Comon et al. 2008]. More formally, a finite tree automaton is defined as follows:

Definition 6.1. **(FTA)** A bottom-up finite tree automaton (FTA) over alphabet Σ is a tuple $\mathcal{A} = (Q, Q_f, \Delta)$ where Q is the set of states, $Q_f \subseteq Q$ are the final states, and Δ is a set of transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow q$ where $q, q_1, \ldots, q_n \in Q$ and $\ell \in \Sigma$.

Following prior work [Wang et al. 2017a,b], the alphabet Σ in our context corresponds to constructs in the underlying programming language; FTA states correspond to a finite set of values (i.e., constants); final states indicate values that satisfy a given specification; and transitions encode the semantics of the programming language. For instance, a transition $+(1, 2) \rightarrow 3$ indicates that adding the integers 1 and 2 yields 3.

Since tree automata accept trees, we view each term over alphabet Σ as a tree $T = (n, V, E)$ where n is the root node, V is a set of labeled vertices, and E is the set of edges. We say that a term T is accepted by an FTA if we can rewrite T to some state $q \in Q_f$ using transitions Δ. Finally, the language of a tree automaton $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$ and consists of the set of all terms accepted by $\mathcal{A}$.

Example 6.2. Consider a tree automaton $\mathcal{A}$ with states $Q = \{q_0, q_1\}$, final states $Q_f = \{q_0\}$, and the following transitions:

$$\Delta = \{x() \rightarrow q_1, \texttt{xor}(q_i, q_i) \rightarrow q_0, \texttt{xor}(q_i, q_j) \rightarrow q_1 \text{ if } i \neq j\}$$

where x has arity zero and `xor` is a binary function. $\mathcal{A}$ accepts boolean equations combining `xor` and x, where the resulting boolean equation evaluates to *false* when x is initially *true*.

Next, we define the notion of an *accepting run* of an FTA:

Definition 6.3. **(Accepting run)** An *accepting run* of an FTA $\mathcal{A} = (Q, Q_f, \Delta)$ is a pair (T, L) where $T = (n, V, E)$ is a term that is accepted by $\mathcal{A}$ and L is a mapping from each node in V to an FTA state such that the following conditions are satisfied:

(1) $L(n) \in Q_f$
(2) If n has children $n_1, \ldots, n_k$ such that $L(n) = q$ and $L(n_1) = q_1, \ldots, L(n_k) = q_k$, then $\mathsf{Label}(n)(q_1, \ldots, q_k) \rightarrow q$ is a transition in Δ.

Example 6.4. Figure 8 shows an accepting run (T, L) over the FTA described in Example 6.2, where T has nodes $\{n_0, n_1, n_2\}$ and edges from n_0 and n_1 to n_2. The labels of n_0 and n_1 are x and the label of n_2 is `xor`. This run is accepting since $L(n_2) = q_0 \in Q_f$.

6.2 Angelic Synthesis Algorithm

In this section, we describe an FTA-based implementation of the SynthesizeAngelic procedure that is invoked at line 2 of Algorithm 1. This procedure, which is summarized in Algorithm 2, takes as input a ground specification χ and returns one of two things: If angelic synthesis is successful, the output is a program P that angelically satisfies χ, together with a witness ω to angelic satisfaction.

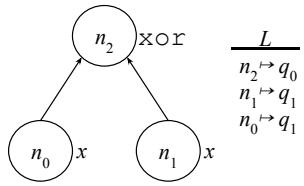

Fig. 8. An example tree accepted by the automaton described in Example 6.2. The the tree and the associated mapping L comprise an accepting run of the FTA.

input: A ground specification χ
output: Success(P, ω) for program P and angelic witness ω; Failure(κ) for anti-specification κ

```
1:  procedure SynthesizeAngelic(χ)
2:      κ ← ∅
3:      for each φ ∈ DNFClauses(χ) do
4:          first ← true
5:          ψ ← true
6:          for each f(v) op v′ ∈ φ do
7:              ψ ← ψ ∧ f(v) op v′
8:              A ← BuildAngelicFTA(v, φ)
9:              if first then A* ← A
10:                 first ← false
11:             else A* ← Intersect(A*, A)
12:             if L(A*) = ∅ then break
13:         if L(A*) ≠ ∅ then
14:             (P, L) ← GetAcceptingRun(A*)
15:             ω ← GetWitness(P, L)
16:             return Success(P, ω)
17:         else κ ← κ ∪ {ψ}
18:     return Failure(κ)
```

Algorithm 2. Angelic synthesis procecdure based on tree automata

On the other hand, if there is no program that angelically satisfies χ, SynthesizeAngelic returns a set of ground formulas κ that serve as an anti-specification satisfying Equation 5.

The algorithm starts by converting the specification χ to disjunctive normal form (DNF) at line 3 and iterates over each of the DNF clauses (line 3–17). If there is a program P that angelically satisfies *any* clause φ, then it returns P (and its corresponding witness ω) as a solution (line 16). On the other hand, if it exhausts all clauses without successfully finding a program, the algorithm returns Failure at line 18.

In more detail, each clause φ is a conjunction of atomic predicates of the form $f(v)$ op v' where v and v' are constants (since χ is a ground specification). The nested loop at lines 6–12 iterates over all of these predicates, builds an FTA $\mathcal{A}$ for each of them (line 8), and constructs a version space $\mathcal{A}^*$ satisfying *all* of them by taking the intersection of all FTAs (line 10). If the language of the resulting automaton $\mathcal{A}^*$ becomes empty (line 12), then this means there is no program satisfying the current clause so the algorithm moves on to the next clause.

$$\textsc{Init}\ \frac{}{q_{v_{in}} \in Q \qquad \mathtt{x}() \to q_{v_{in}} \in \Delta}$$

$$\textsc{Final}\ \frac{q_v \in Q \qquad \mathsf{SAT}(\varphi \wedge f(v_{in}) = v)}{q_v \in Q_f}$$

$$\textsc{Unit}\ \frac{}{q_{\mathtt{unit}} \in Q \qquad \mathtt{unit}() \to q_{\mathtt{unit}} \in \Delta}$$

$$\textsc{Pair}\ \frac{q_{v_1} \in Q \qquad q_{v_2} \in Q}{q_{(v_1,v_2)} \in Q \qquad (\cdot\,,\cdot)(q_{v_1}, q_{v_2}) \to q_{(v_1,v_2)} \in \Delta}$$

$$\textsc{Fst}\ \frac{q_{(v_1,v_2)} \in Q}{q_{v_1} \in Q \qquad \mathtt{fst}(q_{v_1,v_2}) \to q_{v_1} \in \Delta}$$

$$\textsc{Snd}\ \frac{q_{(v_1,v_2)} \in Q}{q_{v_2} \in Q \qquad \mathtt{snd}(q_{(v_1,v_2)}) \to q_{v_2} \in \Delta}$$

$$\textsc{Inl}\ \frac{q_v \in Q}{q_{\mathtt{inl}\ v} \in Q \qquad \mathtt{inl}(v) \to q_{\mathtt{inl}\ v} \in \Delta}$$

$$\textsc{Inr}\ \frac{q_v \in Q}{q_{\mathtt{inr}\ v} \in Q \qquad \mathtt{inr}(v) \to q_{\mathtt{inr}\ v} \in \Delta}$$

$$\textsc{Angelic Recursion}\ \frac{\mathsf{SAT}(\varphi \wedge f(v_1) = v_2) \qquad q_{v_1} \in Q}{q_{v_2} \in Q \qquad \mathtt{f}(q_{v_1}) \to q_{v_2} \in \Delta}$$

$$\textsc{Uneval}\ \frac{}{\bot \in Q}$$

$$\textsc{Uneval Prod}\ \frac{\ell \in \Sigma}{\bot \in Q \qquad \ell(\bot, \ldots, \bot) \to \bot}$$

$$\textsc{Switch Left}\ \frac{q_{\mathtt{inl}\ v_3} \in Q \qquad q_{v_1} \in Q}{\mathtt{switch}(q_{\mathtt{inl}\ v_3}, q_{v_1}, \bot) \to q_{v_1} \in \Delta}$$

$$\textsc{Switch Right}\ \frac{q_{\mathtt{inr}\ v_3} \in Q \qquad q_{v_2} \in Q}{\mathtt{switch}(q_{\mathtt{inr}\ v_3}, \bot, q_{v_2}) \to q_{v_2} \in \Delta}$$

Fig. 9. Inference rules for BuildAngelicFTA(v_{in}, φ).

On the other hand, if the final version space $\mathcal{A}^*$ is non-empty after processing an entire clause (line 13), then we know that there exists a program that satisfies this clause under the angelic semantics. In this case, the algorithm finds an accepting run of this FTA, extracts a witness ω to angelic satisfaction by calling the GetWitness procedure at line 15, and returns "success" at line 16.

Finally, if the algorithm exhausts all DNF clauses without finding a program, it returns Failure at line 18. In particular, the anti-specification κ at line 18 consists of a set of *unsynthesizable cores* (UC), where each UC ψ is a conjunction of predicates such that there is no program P that angelically satisfies ψ. Thus, it is always safe to strengthen the specification using the negation of an unsynthesizable core.

In the remainder of this subsection, we discuss the BuildAngelicFTA and GetWitness procedures in more detail.

6.2.1 FTA Construction using Angelic Semantics. We now explain BuildAngelicFTA procedure that takes as input a value v_{in} and a DNF clause φ and returns an FTA $\mathcal{A}$ whose language is the set of all programs that angelically satisfy φ on input v_{in}. That is:

$$P \in \mathcal{L}(\mathcal{A}) \iff P \models^{😇}_{v_{in}} \varphi$$

This procedure is summarized in Figure 9 using inference rules that stipulate which states and transitions should be part of the constructed FTA. In particular, states in the FTA are of the form q_v where v is a value that arises when angelically executing some program P on input v_{in}. In addition, there is a special state $\bot$ that denotes the value of expressions that are never evaluated on input

v_{in}. Since our language contains conditionals in the form of switch statements, this special state $\perp$ is useful for representing the unknown value of branches that are never evaluated during an execution.

Next, we explain each of the rules from Figure 9 in more detail:

- The first rule, labeled Init, adds the state $q_{v_{in}}$ to the FTA and adds a transition $x() \to q_{v_{in}}$. Since x represents the program input, this rule essentially corresponds to binding x to value v_{in}.
- The next rule, labeled Final, marks the final states of the FTA. In particular, since we want the language of the FTA to be those programs that angelically satisfy φ on input v_{in}, we mark a state q_v as accepting if $f(v_{in}) = v$ is consistent with the given specification φ.
- The next four rules (Unit, ..., Inr) add FTA states and transitions for the different contructors in our language. For example, according to the Pair rule, if q_{v_1} and q_{v_2} are FTA states, then we also add $q_{(v_1,v_2)}$ as a state of the FTA as well as a corresponding transition for the pair constructor.
- The rule labeled Angelic Recursion encodes angelic execution semantics. In particular, if q_{v_1} is an FTA state, then we add a transition $f(q_{v_1}) \to q_{v_2}$ as long as the formula $f(v_1) = v_2$ is consistent with φ.
- The last two rules encode the semantics of switch statements. In particular, there is a transition $\texttt{switch}(q_{v_0}, q_{v_1}, \perp) \to q_{v_1}$ (resp. $\texttt{switch}(q_{v_0}, \perp, q_{v_2}) \to q_{v_2}$) iff the `inl` (resp. `inr`) branch of the switch statement is executed on v_0 and produces value v_1 (resp. v_2). As mentioned earlier, the special state $\perp$ encodes the unknown value of expressions that are not evaluated, and the Uneval Prod rule is used to propagate such "unevaluated" values.

The following theorem states the soundness of our angelic synthesis procedure for a specific input v_{in}:

Theorem 6.5. *If* BuildAngelicFTA(v_{in}, φ) *returns* $\mathcal{A}$*, then* $P \models^{\text{😇}}_{v_{in}} \varphi$ *for every* $P \in \mathcal{L}(\mathcal{A})$.

Proof. The proof is in the full version of the paper. □

The following theorem generalizes this from individual inputs to ground specifications:

Theorem 6.6. *Let* φ *be a ground formula such that:*

$$V = \{v_i \mid f(v_i) \in \mathsf{Terms}(\varphi)\}$$

Then, if BuildAngelicFTA(v_i, φ) *returns* $\mathcal{A}_i$ *for inputs* $V = \{v_1, \ldots, v_n\}$*, then, for every* $P \in \mathcal{L}(\mathcal{A}_1) \cap \ldots \cap \mathcal{L}(\mathcal{A}_n)$*, we have* $P \models^{\text{😇}} \varphi$.

Proof. As the definition of angelic satisfaction simply requires satisfaction on every individual input, this comes directly from Theorem 6.5 and the definition of intersection. □

6.2.2 Finding Witnesses to Angelic Satisfaction. In this section, we describe our procedure for finding witnesses to angelic satisfaction. In particular, given an accepting run (P, L) of the tree automaton where P is a program (represented as an AST) and L is a mapping from AST nodes to FTA states, GetWitness returns a witness ω of the form $\bigwedge_i f(v_i) = v_i'$ that identifies all assumptions made during the angelic execution associated with labeling function L.

Before we explain the rules from Figure 10, we note that L maps each AST node to a tuple $(v_1, \ldots, v_n)$ where each v_i is a value. In particular, while the states for each individual FTA consist of individual values, recall that Algorithm 2 takes the intersection of several FTAs. Thus, after n intersection operations, the states of the FTA correspond n-tuples of the form $(v_1, \ldots, v_n)$.

With this in mind, Figure 10 presents the GetWitness procedure using inference rules that derive judgments of the form $L \vdash P \rightsquigarrow \omega$. The meaning of this judgment is that ω is a witness

$$\frac{\mathsf{Children}(n) = [n'] \qquad L \vdash (n', V, E) \rightsquigarrow \omega' \qquad \begin{array}{c} P = (n, V, E) \\ L(n) = (v_1, \ldots, v_k) \end{array} \qquad \begin{array}{c} \mathsf{Label}(n) = \texttt{f} \\ L(n') = (v'_1, \ldots, v'_k) \end{array}}{L \vdash P \rightsquigarrow \omega' \wedge \bigwedge_{i \in [1 \ldots k]} f(v'_i) = v_i}$$

$$\frac{\mathsf{Label}(n) \neq \texttt{f} \qquad \mathsf{Children}(n) = [n_1, \ldots, n_k] \qquad P = (n, V, E) \qquad \forall i \in [1 \ldots k].\ L \vdash (n_i, V, E) \rightsquigarrow \omega_i}{L \vdash P \rightsquigarrow \wedge_{i \in [1 \ldots k]} \omega_i}$$

Fig. 10. Inference rules describing the GetWitness procedure.

to angelic satisfaction of P in the angelic execution associated with labeling function L. The first rule in Figure 10 deals with recursive invocations of procedure f. In this case, the root node of the AST is a node n labeled with f, and n has a single child n' (since f takes a single argument). Now, suppose that L maps n to the tuple $(v_1, \ldots, v_k)$ and n' to $(v'_1, \ldots, v'_k)$. Such a transition corresponds to the assumption that the recursive call to f returns value v_i on input v'_i. Thus, the resulting witness includes the conjunct $\bigwedge_i f(v'_i) = v_i$. Furthermore, since the argument to f can contain nested recursive calls, this rule also computes a witness ω' for the sub-AST rooted at n' (i.e., $L \vdash (n', V, E) \rightsquigarrow \omega'$). The final witness is therefore the conjunction of ω' and $\bigwedge_i f(v'_i) = v_i$.

The second rule in Figure 10 deals with the scenario where the top-level expression is *not* a recursive call to f. However, since the sub-expressions may contain recursive calls, we recurse down to the children and obtain witnesses for the sub-expressions. The resulting witness is the conjunction of witnesses for all sub-expressions.

7 IMPLEMENTATION

We have implemented our proposed technique in a tool called Burst that is implemented in OCaml. In this section, we discuss some important implementation details and optimizations omitted from the technical development.

7.1 Termination of Synthesized Programs

To ensure that our synthesized programs terminate, our implementation utilizes a well-founded default ordering on our values. In particular, Burst ensures that it generates terminating programs by only permitting recursive calls on values that are *strictly smaller* than the input. If v_{in} is provided as an input, recursive calls to `f` can only be applied to values v when $v \prec v_{in}$. This prevents generating infinite loops like `let rec f(x) = f(x+1)`.

7.2 Finitization of States

Recall that our angelic synthesis technique uses finite tree automata to find a program that satisfies the specification under the angelic semantics. Further, recall that states in the tree automaton correspond to concrete values, of which there may be infinitely many. Similar to prior work [Wang et al. 2017b], our implementation bounds the number of automaton states using a parameter k that controls the number of applications of the inference rules from Figure 9. By default, this parameter is set to 4.

Another complication in our setting is due to the use of angelic recursion in the inference rules in Figure 9. In particular, under the angelic semantics, a recursive call can return any value that satisfies the specification. If the specification is *true*, there are infinitely many concrete values that

satisfy it. Burst gets around this issue by iteratively constructing FTAs from smallest input to largest, and finitizing as it goes.

For example, consider trying to synthesize a program f with the ground specification $(f(0) \geq 0 \wedge f(0) \bmod 2 = 0) \wedge (f(1) \geq 1 \wedge f(1) \bmod 2 = 0)$. The smallest number involved in this ground specification is 0. As zero is the smallest element in the naturals, Burst cannot make any recursive calls, so there is no need to worry about finitizing the outputs of recursive calls. Thus, Burst can create $\mathcal{A}_0$ (the automaton corresponding to input 0), which has final states 0, 2, and 4. Next, when creating $\mathcal{A}_1$ (the automaton corresponding to input 1), Burst only needs to consider the recursive call $f(0)$, which can only return 0, 2 and 4, as these are the only final states of $\mathcal{A}_0$. In general, Burst only constructs $\mathcal{A}_n$ (the automaton on input n) after constructing $\mathcal{A}_0, \ldots, \mathcal{A}_{n-1}$; it then uses their final states to determine the possible values of the recursive calls.

7.3 Program Selection

In general, there may be many programs that satisfy the given specification, and most synthesis algorithms use heuristics to choose which program to return to the user. One of these heuristics is to prefer smaller programs, and, inspired by the effectiveness of this heuristic in prior work [Feser et al. 2015; Lubin et al. 2020; Osera and Zdancewic 2015], Burst also returns the smallest program in terms of AST size. However, to provide such a minimality guarantee, our implementation slightly deviates from the core synthesis procedure shown in Algorithm 1.

In particular, Algorithm 1 makes recursive calls to two distinct strengthened specifications – one to $\chi \wedge \omega$ (line 10) and one to $\chi \wedge \neg\omega$ (line 11). In this algorithm, the call with input $\chi \wedge \neg\omega$ is made only after the call with input $\chi \wedge \omega$ fails. Our actual implementation maintains a priority queue over these specifications sorted according to the size of the minimal solution for the corresponding angelic synthesis problem. It then explores these programs from smallest to largest. Thus, our implementation guarantees that the program returned to the user is the smallest one among those that satisfy the specification.

7.4 Improving the CEGIS Loop

Recall that our technique can perform synthesis from logical specifications by integrating our proposed approach within a CEGIS loop. While the standard CEGIS paradigm only uses ground formulas for inductive synthesis, our approach can actually utilize the original logical specification when performing angelic synthesis. In particular, when deciding which values can be returned by a recursive call, our implementation utilizes the original logical specification as opposed to the weaker ground specifications. For example, suppose that the original specification is $f(x) < x$ and our current counterexamples include 3 and 4 (i.e., ground specification is $f(3) < 3 \wedge f(4) < 4$). While the ground specification does not constrain the output of recursive call $f(2)$, we can use the original specification to constrain the return value of $f(2)$ to be either 0 or 1 (assuming that x is a natural number).

7.5 Optimizations

Our implementation also utilizes a few standard optimizations described in prior synthesis literature. Since it is common to perform type-directed pruning in synthesis [Feser et al. 2015; Osera and Zdancewic 2015], we construct our FTAs to only accept well-typed programs. Inspired by prior work that utilizes eta-long beta-normal form [Frankle et al. 2016; Lubin et al. 2020; Osera and Zdancewic 2015], we also modify our FTA construction rules to only accept such normalized programs.

8 EVALUATION

In this section, we describe a series of experimental evaluations that are designed to answer the following research questions:

RQ1. Is Burst able to effectively synthesize programs from a variety of different specifications?
RQ2. How does Burst compare against prior work in terms of synthesis efficiency and correctness of synthesized programs?
RQ3. How important is it to combine angelic synthesis with specification strengthening?

All experiments described in this section are performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 running macOS Big Sur with a time limit of 120 seconds.

8.1 Benchmarks and Baselines

To answer the research questions listed above, we use Burst to synthesize 45 recursive functional programs from prior work [Lubin et al. 2020; Osera and Zdancewic 2015] and compare it against the following baselines:

(1) **SMyth** [Lubin et al. 2020], which is a top-down type-directed programming-by-example tool. In particular, SMyth generalizes Myth [Osera and Zdancewic 2015] to handle input-output examples that are not trace-complete.
(2) **Synquid** [Polikarpova et al. 2016], which performs synthesis from liquid types.
(3) **Leon** [Kneuss et al. 2013], which is a synthesizer that performs synthesis from logical specifications.

8.2 Specifications

To evaluate whether Burst can handle a variety of different specifications and to compare it against different tools, we consider three classes of specifications for each of our 45 benchmarks:

(1) **IO:** These are input-output examples written by developers of SMyth [Lubin et al. 2020].
(2) **Ref:** These are reference implementations written by us.
(3) **Logical:** These are logical specifications that specify pre- and post-conditions (or, in the case of Synquid, refinement types) on the function to be synthesized.

While Burst can perform synthesis from all three classes of specifications listed above, not all baselines can effectively handle these different specifcations. Thus, we only compare against SMyth on the **IO** and **Ref** specifications and against Leon and Synquid for the **Logical** specifications. Note that SMyth can be adapted to perform synthesis from a reference implementation by integrating it inside a CEGIS loop and obtaining input-output examples from the reference implementation. Furthermore, while Leon and Synquid can, *in principle*, handle **IO** specifications, prior work has shown that they are not effective when used for this purpose [Lubin et al. 2020]. Thus, we only compare against Leon and Synquid on the **Logical** specifications.

8.3 Synthesis from Input/Output Specifications

Figure 11 presents the results of our comparison against SMyth on synthesis tasks from **IO** specifications. Here, the column labeled "Time" shows the synthesis time in seconds, and a cross mark (✗) indicates failure (e.g., time-out). The column labeled "Correct?" shows whether the synthesized program is the one intended by the user. The column labeled "Size" shows the size of the synthesized program. In particular, when synthesis is successful, the returned program always satisfies the provided IO examples, however, it may or may not be the program intended by the user. Thus, this additional column allows us to evaluate how generalizable the synthesis results of these tools are.

Test	Burst			SMyth		
	Time (s)	Correct?	Size	Time (s)	Correct?	Size
bool-always-false	0.02	✓	4	0.04	✓	4
bool-always-true	0.04	✓	4	0.04	✓	4
bool-band	0.04	✓	14	0.04	✓	14
bool-bor	0.04	✓	14	0.04	✓	14
bool-impl	0.04	✓	13	0.03	✓	14
bool-neg	0.03	✓	10	0.02	✓	12
bool-xor	0.04	✓	22	0.04	✓	22
list-append	0.19	✓	31	0.04	✓	24
list-compress	1.12	✓	63	✗	N/A	N/A
list-concat	0.07	✓	21	0.04	✓	21
list-drop	1.90	✓	24	0.08	✓	27
list-even-parity	0.07	✓	32	0.07	✗	36
list-filter	0.82	✓	56	0.12	✗	41
list-fold	0.20	✗	42	1.49	✗	30
list-hd	0.04	✓	12	0.04	✓	13
list-inc	0.07	✓	20	0.07	✓	10
list-last	0.08	✓	26	0.04	✓	27
list-length	0.04	✓	15	0.03	✓	16
list-map	0.08	✓	35	0.49	✗	44
list-nth	1.97	✓	35	0.07	✓	29
list-pairwise-swap	13.10	✓	53	0.31	✓	47
list-rev-append	2.01	✓	24	0.08	✓	24
list-rev-fold	0.07	✓	10	0.07	✓	10
list-rev-snoc	2.22	✓	20	0.04	✓	20
list-rev-tailcall	0.14	✗	45	0.05	✓	24
list-snoc	2.61	✓	36	0.05	✓	28
list-sort-sorted-insert	0.14	✓	20	0.05	✓	20
list-sorted-insert	✗	N/A	N/A	1.64	✓	52
list-stutter	6.44	✓	25	0.04	✓	25
list-sum	0.19	✓	10	0.08	✓	10
list-take	✗	N/A	N/A	0.07	✓	33
list-tl	0.04	✓	11	0.03	✓	13
nat-add	0.26	✓	22	0.04	✓	18
nat-iseven	0.07	✗	18	0.04	✓	22
nat-max	0.33	✓	24	0.13	✓	33
nat-pred	0.04	✓	9	0.03	✓	11
tree-binsert	2.16	✓	90	✗	N/A	N/A
tree-collect-leaves	1.17	✓	29	0.08	✓	28
tree-count-leaves	0.33	✓	24	0.80	✓	29
tree-count-nodes	0.18	✓	24	0.39	✓	24
tree-inorder	10.74	✓	29	0.08	✓	28
tree-map	0.58	✓	47	1.02	✓	58
tree-nodes-at-level	39.69	✓	49	✗	N/A	N/A
tree-postorder	3.97	✓	34	✗	N/A	N/A
tree-preorder	0.29	✓	29	0.13	✓	28

Fig. 11. The results of running Burst and SMyth on the IO benchmark suite. A cross mark under the Time column indicates failure (i.e., either timeout or terminating without finding a solution). Under the "Correct?" column "✓" indicates that the synthesized program is the desired one, and "✗" indicates that the synthesized program matches the IO examples but not the user intent. The column labeled "Size" shows the size of the synthesized program.

As we can see from Figure 11, Burst is able to synthesize a program consistent with the IO examples in all but 2 cases, whereas SMyth fails on 4 benchmarks. For the benchmarks that can be solved by both tools, the running time of both tools is quite fast (a few seconds or less) with the exception of a few outliers.

Finally, the programs synthesized by Burst and SMyth are roughly equal in terms of generalization power: for Burst there are 3 cases where the synthesized program is not the intended one, for SMyth there are 4 such cases.

Failure Analysis. Next, we analyze the two benchmarks that Burst fails on and provide some intuition about why it is unable to solve them. For the benchmarks called list-take and list-sorted-insert, Burst times out because the specification does not in any way constrain the outputs of the many possible recursive calls. This both makes FTA creation quite slow and also causes the algorithm to explore many different strengthenings of the specification.

Result #1: When synthesizing programs from IO specifications, Burst is competitive with SMyth, a state-of-the-art tool for synthesizing recursive programs from input-output examples. In particular, Burst can solve two more benchmarks despite not specializing in IO specifications.

8.4 Synthesis from Reference Implementation

In this section, we use Burst to synthesize programs from reference implementations and compare against SMyth. We incorporate both tools into a CEGIS loop and, for each candidate program, check whether it is equivalent to our reference implementation. If not, we ask the verifier for a concrete counterexample I and obtain its corresponding output O by running the reference implementation on I. We then add (I, O) as a new input-output example and continue this process until the synthesizer program is indeed equivalent to the reference implementation.[2]

The results of this evaluation are shown in Figure 12. In particular, the column labeled "Time" indicates synthesis time in seconds, with ✗ indicating failure as before. The second column labeled "# Iters" shows the number of iterations of the CEGIS loop for those benchmarks that can be synthesized. The third column "Size" shows the size of the synthesized program. As we can see from this figure, Burst successfully solves all but two of the benchmarks with an average synthesis time of 4.49 seconds and 4.37 CEGIS iterations on average. In contrast, SMyth fails to solve 12 of these benchmarks, and it takes an average of 3.07 seconds and 4.52 CEGIS iterations. Overall, we believe these results indicate that Burst is able to deal better with the random examples generated by the verifier compared to SMyth.[3]

Failure Analysis. The reason that Burst fails on tree-nodes-at-level is similar to that for pure IO examples: the verifier returns IO pairs for which the results of the many possible recursive calls are highly under-constrained, resulting in slow FTA construction as well as many strengthening steps. Burst fails on list-rev-tailcall due to our requirement for ensuring termination of synthesized programs (see Section 7.1). Concretely, list-rev-tailcall needs to make a recursive call on `([2],[1])` for input `([1;2],[])`, but our default ordering does not consider `([2],[1])` to be strictly less than `([1;2],[])`.

Result #2: Burst is able to synthesize 96% of the programs from a reference implementation. In contrast, SMyth is only able to synthesize 73%.

[2]We actually use bounded testing instead of verification; however, we manually confirmed that the generated programs are indeed equivalent to the reference implementation.

[3]Recall that the IO examples used in the previous experiment are written by the SMyth developers.

Test	Burst			SMyth		
	Time (s)	# Iters	Size	Time (s)	# Iters	Size
bool-always-false	0.02	0	4	0.02	0	4
bool-always-true	0.02	1	4	0.03	1	4
bool-band	13.39	3	14	0.04	3	14
bool-bor	0.04	4	14	0.04	4	14
bool-impl	0.03	3	13	0.05	3	14
bool-neg	0.03	2	10	0.02	2	12
bool-xor	0.02	2	22	0.05	3	22
list-append	0.54	6	31	0.58	8	24
list-compress	1.55	9	63	✗	N/A	N/A
list-concat	1.38	4	21	1.60	4	21
list-drop	0.69	5	24	✗	N/A	27
list-even-parity	0.19	5	32	0.26	7	37
list-filter	1.20	8	56	✗	N/A	N/A
list-fold	1.21	6	37	✗	N/A	N/A
list-hd	0.29	2	12	0.31	2	13
list-inc	0.55	3	20	0.70	2	10
list-last	0.46	3	26	0.57	5	27
list-length	0.41	4	15	0.47	3	16
list-map	0.82	4	35	✗	N/A	N/A
list-nth	0.51	7	35	0.59	5	29
list-pairwise-swap	0.53	6	42	✗	N/A	N/A
list-rev-append	14.55	5	24	✗	N/A	N/A
list-rev-fold	1	2	10	✗	N/A	N/A
list-rev-snoc	6.94	4	20	✗	N/A	N/A
list-rev-tailcall	✗	N/A	N/A	0.68	9	24
list-snoc	0.62	3	36	0.68	7	27
list-sort-sorted-insert	2.12	5	20	1.98	6	20
list-sorted-insert	1.57	6	63	14.22	12	72
list-stutter	1.42	3	25	0.59	4	25
list-sum	1.03	2	10	1.02	2	10
list-take	17.12	9	40	0.63	7	33
list-tl	0.31	2	11	0.27	2	13
nat-add	0.12	6	22	0.13	6	18
nat-iseven	0.03	4	20	0.04	4	21
nat-max	2.28	5	24	0.24	7	42
nat-pred	0.03	2	9	0.03	2	11
tree-binsert	22.10	8	90	✗	N/A	N/A
tree-collect-leaves	9.09	5	29	8.29	5	28
tree-count-leaves	23.57	4	24	8.63	4	29
tree-count-nodes	8.48	4	24	8.25	4	24
tree-inorder	13.98	5	29	11.72	6	28
tree-map	8.98	5	47	✗	N/A	N/A
tree-nodes-at-level	✗	N/A	N/A	26.75	5	38
tree-postorder	21.94	8	34	✗	N/A	N/A
tree-preorder	11.72	4	29	11.89	5	28

Fig. 12. The results of running Burst and SMyth on the **Ref** benchmark suite. A cross mark under the Time column indicates failure (i.e., either timeout or terminating without finding a solution). The column labeled "# Iters" shows the number of iterations within the CEGIS loop. The column labeled "Size" shows the size of the synthesized program.

8.5 Synthesis from Logical Specifications

Figure 13 shows the results of our evaluation on logical specifications for each of Burst, Synquid, and Leon. As before, the column labeled "Time" shows the synthesis time for each tool, and the column labeled "Correct?" shows whether the tools were able to generate the intended program from the given specification. The column "Size" shows the size of the synthesized program.[4] As we can see in this table, Burst solves more benchmarks than Leon and significantly more compared to Synquid. In what follows, we explain the failure cases of each tool and contrast them with each other.

Failure Analysis for Burst. The two unique failure cases for Burst are nat-add and nat-max. For the first one, given the unary encoding of two natural numbers, the goal is to add them, and for the second one, the goal is to return the maximum. While these benchmarks look very easy at first glance, Burst fails on them because the specification is very under-constrained. For example, the specification of nat-max only states that the output should be greater than or equal to both inputs, but under the angelic semantics of recursion, this results in many possible outputs of the recursive calls and causes a blow-up. This is a common theme for Burst across all types of specifications: Since it constructs a version space based on angelic semantics, synthesis becomes more difficult when the specification is "loose" for arguments used in recursive calls.

Behavior of Synquid. At first glance, Synquid seems to perform surprisingly poorly on these benchmarks. Upon further investigation, we found that Synquid is only able to successfully synthesize programs from highly stylized specifications. For example, consider our specification for the list-compress benchmark shown in Figure 14. While Synquid is unable to synthesize a program from this specification within the given time limit, it *can* synthesize the desired program from the specification shown in Figure 15. These specifications are semantically equivalent; however, Synquid's behavior on them is very different. Thus, while it may be possible to re-engineer our specifications so that Synquid performs successful synthesis, coming up with specifications that are Synquid-friendly is a highly non-trivial task.

Comparisons to Leon. Leon performs better than Synquid for our specifications; however, it solves 34 benchmarks compared to the 41 of Burst. Overall, Leon tends to perform relatively poorly on benchmarks with nontrivial branching (e.g., list-compress and tree-binsert). This behavior is likely due to their condition abduction procedure failing to infer the correct branch conditions when they are deeply nested. On the higher-order benchmarks (list-filter, list-map), Leon either reports an error or returns a wrong solution that does not satisfy the provided logical specification.

Result #3: Burst is able to synthesize 91% of the benchmarks from logical specifications and solves more benchmarks than both Leon (76%) and Synquid (49%).

8.6 Ablation Study for Specification Strengthening

Our proposed synthesis algorithm combines angelic synthesis with specification strengthening. However, an alternative approach is to perform enumerative search over all programs that angelically satisfy the specification. That is, one could repeatedly sample solutions to the angelic synthesis problem and test whether they satisfy the specification until we exhaust the search space or find the correct program. In this section, we perform an ablation study to evaluate the benefit of specification strengthening compared to a simpler enumerative search baseline.

The results of this ablation study are presented in Figure 16, where Burst† is a variant of Burst that performs basic enumerative search instead of backtracking search with specification

[4] Leon did not seem to provide an automated way to identify size, so we did not include a "Size" column for it.

Test	Burst			Leon		Synquid		
	Time (s)	Correct?	Size	Time (s)	Correct?	Time (s)	Correct?	Size
bool-always-false	0.08	✓	4	0.04	✓	0.30	✓	3
bool-always-true	0.03	✓	4	0.07	✓	0.31	✓	3
bool-band	0.03	✓	14	6.57	✓	0.32	✓	6
bool-bor	0.05	✓	14	6.62	✓	0.30	✓	6
bool-impl	0.04	✓	13	6.61	✓	0.31	✓	6
bool-neg	0.03	✓	10	1.63	✓	0.29	✓	5
bool-xor	0.04	✓	22	3.60	✓	0.31	✓	9
list-append	0.88	✓	31	44.28	✓	0.51	✓	39
list-compress	9.61	✓	63	✗	N/A	✗	N/A	N/A
list-concat	2.28	✓	21	11.57	✗	0.39	✓	71
list-drop	1.13	✓	33	51.04	✓	0.58	✓	35
list-even-parity	0.70	✓	32	42.82	✓	✗	N/A	N/A
list-filter	1.51	✗	46	✗	N/A	✗	N/A	N/A
list-fold	3.44	✓	37	✗	N/A	✗	N/A	N/A
list-hd	0.32	✓	12	3.82	✓	0.29	✓	21
list-inc	1.68	✓	20	5.30	✓	0.48	✓	64
list-last	0.43	✓	26	12.30	✓	✗	N/A	N/A
list-length	0.40	✓	15	8.45	✓	✗	N/A	N/A
list-map	2.87	✓	35	✗	N/A	✗	N/A	N/A
list-nth	1.13	✓	35	✗	N/A	✗	N/A	N/A
list-pairwise-swap	0.90	✓	53	37.27	✗	✗	N/A	N/A
list-rev-append	1.53	✗	22	12.87	✓	✗	N/A	N/A
list-rev-fold	1.97	✓	10	13.87	✓	✗	N/A	N/A
list-rev-snoc	1.43	✗	18	13.57	✗	✗	N/A	N/A
list-rev-tailcall	✗	N/A	N/A	✗	N/A	✗	N/A	N/A
list-snoc	0.87	✗	33	27.68	✓	0.40	✓	54
list-sort-sorted-insert	11.79	✓	20	7.53	✓	✗	N/A	
list-sorted-insert	4.78	✓	63	✗	N/A	✗	N/A	N/A
list-stutter	3.11	✓	25	8.36	✓	0.44	✓	42
list-sum	1.04	✓	10	12.15	✓	0.44	✓	72
list-take	11.02	✗	37	✗	✗	0.47	✓	39
list-tl	0.36	✓	11	2.55	✓	0.30	✓	10
nat-add	✗	N/A	N/A	19.38	✓	✗	N/A	N/A
nat-iseven	0.03	✓	20	9.11	✓	0.34	✓	21
nat-max	✗	N/A	N/A	25.48	✗	0.56	✓	26
nat-pred	0.02	✓	9	2.77	✓	0.31	✓	16
tree-binsert	✗	N/A	N/A	✗	N/A	✗	N/A	N/A
tree-collect-leaves	8.29	✓	29	9.01	✓	✗	N/A	N/A
tree-count-leaves	8.16	✓	24	5.71	✓	9.77	✗	75
tree-count-nodes	13.49	✓	24	5.31	✓	16.26	✗	71
tree-inorder	64.82	✓	29	5.86	✓	✗	N/A	N/A
tree-map	11.79	✓	47	✗	N/A	✗	N/A	N/A
tree-nodes-at-level	42.73	✓	49	✗	N/A	✗	N/A	N/A
tree-postorder	23.21	✓	34	10.42	✓	✗	N/A	N/A
tree-preorder	18.99	✓	29	8.12	✓	✗	N/A	N/A

Fig. 13. The results of running Burst, Synquid, and Leon on the **Logical** benchmark suite. The result "✗" under "Time" indicates failure (in this case, timeout). Under the "Correct?" column,"✓" (resp. "✗") indicates that the synthesized program was (resp. not) the intended one. The column labeled "Size" shows the size of the synthesized program.

```
data List where
  Nil :: List
  Cons :: Nat -> List -> List

measure heads :: List -> Set Nat where
  Nil -> []
  Cons x xs -> [x]

measure no_adjacent_dupes :: List -> Bool where
  Nil -> True
  Cons x xs -> !(x in heads xs) && no_adjacent_dupes xs

compress :: xs: List a -> {List | elems xs == elems _v && no_adjacent_dupes _v}
compress = ??
```

Fig. 14. Our list-compress benchmark specification for Synquid.

```
data PList a <p :: a -> PList a -> Bool> where
        Nil :: PList a <p>
        Cons :: x: a -> xs: {PList a <p> | p x _v} -> PList a <p>

measure heads :: PList a -> Set a where
  Nil -> []
  Cons x xs -> [x]

type List a = PList a <{True}>
type CList a = PList a <{!(_0 in heads _1)}>

compress :: xs:List a -> {CList a | elems xs == elems _v}
compress = ??
```

Fig. 15. Alternative list-compress benchmark specification for Synquid.

	IO	Ref	Logical
Burst	96%	96%	91%
Burst†	73%	78%	84%

Fig. 16. Number of benchmarks that can be solved within the time limit for each of the three specifications.

strengthening. As we can see from this table, Burst with specification strengthening solves more benchmarks within the given time limit, and this difference is particularly pronounced for the **IO** and **Ref** specifications. For **Logical** specifications, the difference between Burst and Burst† is less stark due to the optimization described in Section 7.4.

Result #4: The variant of Burst that performs enumerative search over angelic synthesis results solves fewer benchmarks than Burst (with specification strengthening) for all three specification types.

9 RELATED WORK

The prior work most related to this paper can be divided into four overlapping categories: (i) bottom-up synthesis, (ii) version-space-based synthesis, (iii) synthesis of functional recursive programs, and (iv) synthesis based on angelic semantics. Now we elaborate on these categories of work. For a broader survey of program synthesis, see Gulwani et al. [2017].

Bottom-up Synthesis. Bottom-up enumeration is a classic approach to program synthesis. A canonical example is Transit [Udupa et al. 2013]. Transit grows a pool of programs of increasing complexity, ensuring that no program in the pool is *observationally equivalent* to another program in the pool. The Stun [Alur et al. 2015] approach generalizes this method by decomposing the input-output specification into multiple parts, synthesizing programs that work for these sub-specifications in a bottom-up way, then combining these programs using a form of anti-unification. Bustle [Odena et al. 2020] offers another generalization, using a learning algorithm to guide bottom-up exploration. However, none of these methods handle programs with general recursion.

Escher [Albarghouthi et al. 2013] is a bottom-up inductive synthesis approach that handles recursion. The algorithm here combines a forward search, in which terms are generated bottom-up, with a procedure for inferring conditional statements. However, a key limitation of this approach is that it requires a trace-complete specification to handle recursive calls.

Version-Space-Based Synthesis. Version space approaches to synthesis use an efficient data structure to represent the set of all programs that satisfy a specification. Early techniques [Lau et al. 2003] proved hard to scale. FlashFill [Gulwani 2011], which represented version spaces using a form of e-graphs [Downey et al. 1980], was a major leap forward. FlashFill's success led to followup methods, for example, FlashExtract [Le and Gulwani 2014], FlashRelate [Barowy et al. 2015], and Refazer [Rolim et al. 2017]. This line of work culminated in FlashMeta [Polozov and Gulwani 2015], a framework for version-space-based synthesis that supports the above methods as instantiations. Unlike Burst, these methods all construct versions spaces top-down. They work backward from the desired output for a specific input, iteratively producing subgoals describing the unknown parts of the target program, and then construct version spaces for these subgoals.

The use of tree automata (FTAs) as a version space representation was first explored by Madhusudan [2011] in the setting of reactive synthesis. Subsequently, the Dace [Wang et al. 2017b], Relish [Wang et al. 2018], and Blaze [Wang et al. 2017a] systems used tree automata to represent version spaces in the synthesis of functional programs. Like Burst, these three approaches proceed bottom-up: rather than starting from the desired outputs and producing subgoals for incomplete programs, they start from the input values and propagate these inputs through a space of programs. However, for reasons explained earlier, these methods cannot handle general recursion.

Synthesis of Recursive Programs. The synthesis of (functional) recursive programs has a long history. Most methods in this area consider synthesis from examples, but synthesis from richer specifications, such as refinement types, has also been considered. The Thesys [Summers 1977] and Igor2 [Kitzelmann et al. 2006] systems are two early examples of work of this sort. Given a set of examples, these methods first synthesize straight-line programs in a top-down manner, then identify patterns within a given program, then generalize these patterns into a recursive program.

More recently, the Myth [Osera and Zdancewic 2015] and λ^2 [Feser et al. 2015] systems introduced types as a means of directing an inductive synthesis process. Myth2 [Frankle et al. 2016] extended Myth with more complex types of refinement types, including negative examples, intersection, and union types. All these approaches are top-down; also, all of them, except for λ^2, rely on trace-completeness to handle recursive calls. While λ^2 does not assume trace-completeness, it only applies

deductive reasoning to limited forms of recursion assuming trace-completeness, and defaults to brute-force enumeration when handling general recursion or on non-trace-complete examples.

SMyth [Lubin et al. 2020] is a generalization of Myth that also performs top-down deduction-aided search, but does not have the trace-completeness requirement. To handle non-trace-complete specifications, SMyth generates partial programs, then propagates constraints from partial programs to the remaining holes. This propagation is both complex and domain-specific, and incomplete. By contrast, Burst relies on a single generalizable principle of angelic recursion, and is complete.

Leon [Kneuss et al. 2013] performs synthesis modulo recursive functions. Leon takes a pre-/post-condition specification and searches for a recursive function that can satisfy it. Leon solves this task using a term generation engine that produces candidate programs, a condition abduction engine that synthesizes branches, and a verification engine that evaluates candidate solutions. Leon's term generation runs similarly (though not exactly, due to the lack of angelic execution) to our Burst† ablation; it does not search through programs based on recursive results but simply based on increasing cost. Its condition abduction engine is top-down and quite distinct from ours.

Synquid [Polikarpova et al. 2016] synthesizes programs from polymorphic refinement types. Such types are expressive and allow the specification of desired functions in a way that is both compositional and tight. However, while Synquid can synthesize nearly any function from a carefully crafted refinement type, there are many kinds of realizable specifications on which it simply gives up. In particular, Synquid cannot synthesize from specifications that are non-inductive, including many of the specifications in our benchmark suite. More generally, Burst and Synquid address different problems: Synquid focuses on always being able to synthesize from a well-written refinement type, while Burst focuses on best-effort synthesis from arbitrary logical specifications.

The recent Cypress [Itzhaky et al. 2021] system targets synthesis of recursive programs from separation logic specifications. Cypress generates a satisfying straight-line program, then "folds" that program into a generalized recursive procedure. We attempted a similar strategy in our setting but found the space of possible foldings to be prohibitively large. In contrast, our synthesis algorithm follows a lazier strategy: instead of synthesizing the full search space, then finding ways to fold it together, we overapproximate the search space and then discover ways to refine it.

Angelic Synthesis. While angelic non-determinism has been used to expose synthesizers to programmers, there is almost no work on the use of angelic semantics as a core part of a synthesis algorithm. The one approach that we know of is FrAngel [Shi et al. 2019], which adds control structures to the well-studied problem of component-based synthesis. FrAngel first identifies candidate partial programs by synthesizing programs with *no* control structures, but instead with *angelic placeholders*, then attempts to place appropriate control structures in place of the angelic placeholders. In contrast, Burst does not have angelic placeholders but instead updates the generated code itself to fulfill the requirements put in place by angelic recursion.

Angelic execution has used in the related field of program repair. SPR [Long and Rinard 2015] uses angelic executions to identify candidate locations for condition repair, then instantiates those conditions in a second phase. SPR is similar to FrAngel, as it stages the synthesis into a sketch identification stage (using Angelic Semantics to identify promising sketches), and a sketch completion stage. Angelix [Mechtaev et al. 2016] generalizes this approach to perform multi-location repairs, through their novel "angelic forest" data structure.

10 FUTURE WORK

As found in our failure analysis in Section 8, Burst has issues with severely underconstrained specifications. This is due to two primary reasons: (1) extensive backtracking and (2) output blowup. To address (1) we believe that additional work in anti-specifications could be helpful. By identifying

a more general anti-specification, more parts of the search space are eliminated, necessitating less backtracking. To address (2) we believe that integrating an abstraction refinement algorithm like that used by Blaze [Wang et al. 2017a] could tame blowups in candidate outputs.

DryadSynth [Huang et al. 2020] and Duet [Lee 2021] have shown that integrating bottom-up and top-down approaches results in synthesizers greater than the sum of their parts, and we believe these findings would generalize to problems involving recursion. In particular, we think there is promising future work in integrating Burst with a top-down recursive synthesizer like SMyth.

Lastly, there is a large class of important specifications that Burst cannot currently address – relational specifications. Relational specifications describe the interactions between multiple different program runs. For example, this means that Burst cannot synthesize programs that are idempotent, as we cannot reduce the postcondition $f(f(x)) = f(x)$ to a ground specification. We think there is promising future work in integrating Burst with existing techniques for synthesizing programs from relational specifications [Wang et al. 2018].

11 CONCLUSION

In this paper, we presented a new technique for synthesizing recursive functional programs. Our approach differs from prior work in this space as it performs synthesis in a *bottom-up* fashion. Our algorithm first performs *angelic synthesis* wherein recursive calls may return any value consistent with the specification. This result may be spurious, so our method analyzes the assumptions made in angelic executions and gradually strengthens the specification to find the correct program.

We have implemented the proposed algorithm in a tool called Burst and showed that it can synthesize programs from a variety of specifications, including examples, reference implementations, and logical formulas. Our comparison against three synthesizers (SMyth, Leon, and Synquid) shows that Burst advances the state-of-the-art in synthesizing recursive functional programs.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers, our anonymous shepherd, Ben Mariano, and Todd Millstein for their helpful feedback. We thank Michael James, Tristan Knoth, and Nadia Polikarpova for their help with Synquid and the tooling surrounding it. This work is supported in part by NSF Award 1762299, NSF Award 1811865, NSF Award 1918651, DARPA Contract FA8750-20-C-0208, and US Air Force and DARPA Contract FA8750-20-C-0002.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. *SIGPLAN Not.* 50, 6 (jun 2015), 218–228. https://doi.org/10.1145/2813885.2737952

M. Broy and M. Wirsing. 1981. On the algebraic specification of nondeterministic programming languages. In *CAAP '81*, Egidio Astesiano and Corrado Böhm (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 162–179. https://doi.org/10.1007/3-540-10828-9_61

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. *Tree Automata Techniques and Applications.* 262 pages. https://hal.inria.fr/hal-03367725

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the Common Subexpression Problem. *J. ACM* 27, 4 (Oct. 1980), 758–771. https://doi.org/10.1145/322217.322228

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *SIGPLAN Not.* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. *SIGPLAN Not.* 51, 1 (Jan. 2016), 802–815. https://doi.org/10.1145/2914770.2837629

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* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and TrendsÂ® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

Emanuel Kitzelmann, Ute Schmid, Roland Olsson, and Leslie Pack Kaelbling. 2006. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research* 7, 2 (2006).

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

Tessa Lau, Pedro Domingos, and Daniel S. Weld. 2003. Learning Programs from Traces Using Version Space Algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture* (Sanibel Island, FL, USA) *(K-CAP '03)*. Association for Computing Machinery, New York, NY, USA, 36–43. https://doi.org/10.1145/945645.945654

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

Woosuk Lee. 2021. Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 54 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434335

Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408991

Parthasarathy Madhusudan. 2011. Synthesizing Reactive Programs. In *Computer Science Logic (CSL'11) - 25th International Workshop/20th Annual Conference of the EACSL (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 12)*, Marc Bezem (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 428–442. https://doi.org/10.4230/LIPIcs.CSL.2011.428

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2021. Bottom-up Synthesis of Recursive Functional Programs using Angelic Execution. arXiv:2107.06253 [cs.PL]

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, and Charles Sutton. 2020. BUSTLE: Bottom-up program-Synthesis Through Learning-guided Exploration. *arXiv preprint arXiv:2007.14381* (2020).

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

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* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (jan 2019), 29 pages. https://doi.org/10.1145/3290386

Phillip D. Summers. 1977. A Methodology for LISP Program Construction from Examples. *J. ACM* 24, 1 (jan 1977), 161–175. https://doi.org/10.1145/321992.322002

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017a. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158151

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133886

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276525