Relational Synthesis of Recursive Programs via Constraint Annotated Tree Automata

Anders Miltner[1], Ziteng Wang[2], Swarat Chaudhuri[2], and Isil Dillig[2]

[1] Simon Fraser University, Canada
miltner@cs.sfu.ca
[2] University of Texas at Austin, USA
{ziteng,swarat,isil}@cs.utexas.edu

Abstract. In this paper, we present a new synthesis method based on the novel concept of *constraint annotated tree automaton (CATA)*. A CATA is a variant of a finite tree automaton (FTA) where the acceptance of a term by the automaton is conditioned upon the logical satisfiability of a formula. In the context of program synthesis, CATAs allow the construction of a more precise version space than FTAs by ruling out programs that make inconsistent assumptions about the unknown semantics of functions under synthesis. We apply our proposed algorithm to synthesizing recursive (or mutually recursive) procedures from relational specifications and demonstrate that our method allows solving synthesis problems that are beyond the scope of existing approaches.

1 Introduction

Program synthesis, the task of automatically generating programs that meet a given specification, has found numerous applications, including both user-facing domains like data science [15, 16, 47, 48] as well as software engineering tasks [21, 32, 36, 37, 40]. Program synthesizers can be classified among two dimensions, namely (1) whether they target a domain-specific or general programming language, and (2) what type of specification they require. Many synthesizers targeting end-users utilize domain-specific languages and only require informal specifications such as input-output examples or natural language [4, 9, 20, 52]. In contrast, program synthesizers targeting developers tend to require formal specifications and need to handle a richer set of language features [33, 39, 41, 49].

In the context of synthesizing general-purpose programs from logical specifications, two aspects have proven to be particularly challenging:

- **Recursion:** Problems that require synthesizing recursive, or mutually recursive, functions have proven to be particularly difficult to solve. Despite recent progress in this area [33, 53], synthesizers that tackle recursive functions are not as effective as those that target domain-specific languages.
- **Relational specifications:** With the exception of one prior research effort [51], most synthesizers do not handle *relational specifications*. However, in practice, relational specifications are particularly relevant: for example,

parametrized unit tests [13, 44] and *property-based tests* [11, 18, 28], which are becoming increasingly more popular, are, in essence, relational specifications.

While prior research has tried to tackle each of these problems in isolation, there is no prior work that has attempted to solve synthesis problems that involve *both* recursive procedures and relational specifications. In this paper, we ask the question, "Is it possible to synthesize recursive, or even mutually recursive, functions from relational specifications?" For example, given the specification:

$$\mathsf{even}(0) \land \forall x.\ (\mathsf{even}(x) \Leftrightarrow \neg\mathsf{odd}(x) \land \mathsf{even}(x) \Rightarrow \mathsf{odd}(x+1))$$

can we generate correct implementations of both `even` and `odd`? This task is quite difficult, as it requires simultaneously solving the challenges introduced by recursion and relational specifications. Intuitively, handling recursion is hard because the synthesizer does not know the semantics of terms that involve recursive calls to the function being synthesized. Similarly, relational specifications pose a significant challenge because such specifications do not constrain the input-output behavior of any *individual function*. For these reasons, the search space for the underlying synthesis problem becomes enormous, and standard techniques that facilitate search space pruning become insufficient.

Interestingly, some of the prior research efforts [33, 51] on relational and recursive synthesis adopt roughly the same solution: they construct a finite tree automaton (FTA) whose language *over-approximates* the space of programs consistent with the specification. The key idea is to allow *non-deterministic* FTA transitions that encode uncertainty about the semantics of the function being synthesized. However, since the resulting version space is over-approximate, these techniques need to combine FTA construction with backtracking search.

Given the similarity between these two techniques that address two (apparently orthogonal) challenges, one might be tempted to ask: *"Can we use the same idea to solve synthesis problems that involve both relational specifications* and *that also require synthesizing recursive procedures?"* In principle, the answer to this question is "yes"; however, as we show experimentally, the resulting technique does not yield an effective solution in practice.

The main contribution of this paper is a more effective approach for synthesizing recursive programs from relational specifications. Our method is based on the novel concept of *constraint annotated tree automaton (CATA)*, a new type of FTA where non-deterministic transitions are constrained by logical formulae in a first-order theory. Similar to a standard FTA, a necessary condition for accepting a tree T is to find a run of the automaton on T that ends in an accepting state. However, because transitions of a CATA are only valid under certain conditions, a run of the automaton also induces an *acceptance constraint*, which must be logically satisfiable in order for that run to be valid. Intuitively, CATAs offer a more effective synthesis methodology because their acceptance condition allows us to build an *exact*, rather than *over-approximate*, version space with acceptable overhead. Furthermore, by leveraging an SMT solver to check the acceptance condition of the CATA, we can avoid the need for explicit backtrack-

ing search and can instead piggyback on all research results underlying modern SMT solvers.

In addition to proposing the concept of CATAs and showing how they can be used for synthesis, another key contribution of this paper is a *goal-directed* approach for CATA construction that exploits the specific problem instance at hand. In particular, a naive synthesis approach based on CATAs would require constructing multiple CATAs for different sub-terms in the specification and then taking their intersection. However, as is the case with any type of automaton, intersection is an expensive operation, so a synthesis algorithm that requires many intersections is unlikely to scale. Our method addresses this problem by proposing a more efficient algorithm that minimizes the number of automaton intersections required to create a precise version space.

We have implemented our proposed approach in a new tool called CONTATA and evaluate it on a suite of synthesis benchmarks involving recursion and specified by a relational specification. Our evaluation shows the advantages of our approach over prior techniques that first build a non-deterministic FTA and then perform backtracking search.

2 Motivating Example

In this section, we give an overview of the technique on an extended example where the goal is to automatically synthesize two functions:

```
evens : list a -> list a      odds : list a -> list a
```

Given a list, the `evens` function is expected to return all elements at even indices, and the `odds` function should return elements at odd indices. For example, we have `evens([3,8,2]) = [3,2]` and `odds([3,8,2]) = [8]`. The code for `evens` and `odds` is given below:

```
evens(x) = match x with         odds(x) = match x with
| [] -> []                      | [] -> []
| h:t -> h:odds(t)              | _:t -> evens(t)
```

Note that the `evens` and `odds` functions are mutually recursive: The `evens` function starts by extracting the head of the list, then prepends it to the result of calling `odds` on the tail. The `odds` function skips the first element in its input list, and merely returns the result of calling `evens` on the tail.

Specifying the task. Since our goal is to automatically synthesize these functions, we first need a specification for this task. For the purposes of this example, suppose that the user provides the following specification:

$$\forall x.\forall xs.\ \mathsf{evens}(xs) = \mathsf{odds}(x : xs) \wedge \forall x.\forall y.\forall z.\mathsf{evens}([x, y, z]) = [x, z]$$

The first conjunct describes the *relationship* between evens and odds, namely, that the result of calling odds on $x : xs$ should be the same as calling on evens

on xs. The second part of the specification provides a *symbolic* input-output example. Note that such relational specifications naturally arise in many contexts, including data structure specifications [37, 36], parametrized unit tests [13], and 2-safety properties like commutativity [42].

Prior work. Before describing our technique, we first briefly explain how prior work deals with relational specifications and recursion. First, let us assume that the universal quantifiers in the specification have been instantiated via a standard *counterexample-guided inductive synthesis (CEGIS)* loop. In particular, suppose we have the following *ground formula* during some iteration of CEGIS:

$$\mathsf{evens}(0 : [1, 2]) = \mathsf{odds}([1, 2]) \wedge \mathsf{evens}([0, 1, 2]) = [0, 2] \tag{1}$$

Given such a specification φ for counterexamples $I_1, \ldots, I_n$, a common theme behind prior work [49, 50, 33] is to construct a *version space* in the form of a *finite tree automaton (FTA)* that represents the space of all programs (up to a bounded depth) that are consistent with φ. Specifically, these techniques construct an automaton $\mathcal{A}_i$ for each counterexample I_i and then use FTA intersection to handle the set of all counterexamples. Finally, states that satisfy the specification φ are marked as final, so any tree accepted by the resulting FTA is a solution.

Constructing such an FTA is straightforward when the semantics of all expressions are known: Since the automaton states represent *constants* and the transitions correspond to operators/functions, we can simply add new transitions and nodes to the FTA using the operational semantics of the language. As an example, consider a DSL operator f that takes as input two integers x and y and produces $2x + y$, and let q_x be the automaton state representing constant x. Since the semantics of f are known, the FTA would contain the transition $f(q_1, q_2) \to q_4$ since $f(1, 2)$ is equal to 4.

Unfortunately, recursive procedures and relational specifications pose a significant challenge for FTA construction: When synthesizing a recursive function f, the implementation can recursively call f, but the semantics of f are not yet known, as f is currently under construction. Similarly, when dealing with relational specifications, the implementation of a function f could call another function g, but the semantics of g are also not yet known. To make matters worse, relational specifications constrain the *joint* behavior of multiple functions, so, when constructing the FTA for an *individual* function, we cannot even determine which FTA states should be marked as final.

Limitations of prior work. Existing techniques [33, 51] deal with this challenge by adding *non-deterministic transitions* to the FTA. In particular, given FTA states representing values $c_1, \ldots, c_n$ and an n'ary function f that has yet to be synthesized, the idea is to add a transition of the form $f(c_1, \ldots, c_n) \to c$ as long as the formula $f(c_1, \ldots, c_n) = c$ is *consistent* with the specification. However, since there are many such output values c that are consistent with the specification, this introduces a high degree of non-determinism. Furthermore, relational specifications also introduce non-determinism with respect to final states, so the resulting FTA is very *over-approximate* — that is, the ground

truth program is accepted by the FTA, but not every program accepted by the FTA satisfies the specification. As a result, existing techniques [51, 33] combine FTA construction with backtracking search to look for a valid solution. However, if the synthesis task involves *both* recursion *and* relational specifications, the resulting FTA becomes *so* over-approximate that performing backtracking search over this space of programs is no longer feasible.

Insight behind our approach. Our approach is motivated by the following observation about the shortcoming of prior techniques: *Many programs accepted by the over-approximate FTA make inconsistent assumptions about the unknown semantics of functions being synthesized.* For example, consider the following incorrect solution for the `evens` function for our running example:

```
evens(x) = match x with
| [] -> []
| h:t -> odds(t)++odds(t)
```

This program must be incorrect with respect to the specification (Equation 1) even if we know *absolutely nothing* about the implementation of `odds`: The only way this program can return `[0,2]` on input list `[0,1,2]` is if the first call to `odds` on `[1,2]` returns `[0]` and the second call returns `[2]` on the same input list. But, assuming that `odds` is deterministic, this is clearly infeasible, as we cannot have `odds([1,2]) = [0]` and `odds([1,2]) = [2]` at the same time!

However, prior techniques construct a version space that includes this spurious program: Since the specification does not constrain the behavior of `odds([1,2])` in any way, they would allow *any* transition from `odds([1,2])` to any possible automaton state, including both `[0]` as well as `[2]`. Unfortunately, this leads to many spurious programs, including the “obviously wrong” implementation of `evens` from above.

In this paper, we show how to construct the version space in such a way that such *inconsistent programs* are never part of it. Our key idea is to qualify transitions in the FTA by *logical formulas* that indicate necessary conditions for a transition to be valid. We refer to such an FTA as a *Constraint Annotated Tree Automaton (CATA)* due to presence of constraints on its transitions. Then, a given tree will *only* be accepted by the CATA if there exists a run of the CATA that *both* ends in an accepting state *and* does *not* make inconsistent assumptions. However, because the `evens` implementation above makes inconsistent assumptions about the I/O behavior of `odds`, our approach can immediately rule it out.

3 Preliminaries

A *finite tree automaton* is a type of state machine that accepts trees rather than strings. More formally, FTAs are defined as follows:

Definition 1. (FTA) *A (bottom-up) finite tree automaton (FTA) over a finite alphabet Σ is a tuple $\mathcal{A} = (Q, Q_f, \Delta)$ where Q is a finite set of states, $Q_f \subseteq Q$*

is a set of final states, and Δ is a set of transitions (rewrite rules) of the form $f(q_1, \ldots, q_n) \to q$ where $q, q_1, \ldots, q_n \in Q$ and $f \in \Sigma$.

Each symbol in the alphabet Σ has an arity (rank), and terms of arity k are denoted Σ_k. Each ground term t can be represented in terms of its syntax tree (n, V, E) with root node n, vertices V, and edges E; hence, we use "tree" and "term" interchangeably. We say that a tree t is accepted by an FTA if we can rewrite t to some state $q \in Q_f$ using transitions Δ. The language of an FTA $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, includes all ground terms that $\mathcal{A}$ accepts.

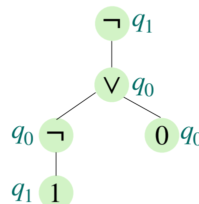


Fig. 1: Tree representing $\neg(\neg 1 \vee 0)$.

Example 1. Consider the FTA $\mathcal{A}$ with states $Q = \{q_0, q_1\}$, $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{\neg\}$, $\Sigma_2 = \{\vee\}$, final states $Q_f = \{q_1\}$, and the transitions Δ:

$$
\begin{array}{llll}
1 \to q_1 & 0 \to q_0 & \vee(q_0, q_0) \to q_0 & \vee(q_0, q_1) \to q_1 \\
\neg(q_0) \to q_1 & \neg(q_1) \to q_0 & \vee(q_1, q_0) \to q_0 & \vee(q_1, q_1) \to q_1
\end{array}
$$

This FTA accepts propositional logic formulas that evaluate to *true*. For instance, Figure 1 shows the tree for formula $\neg(\neg 1 \vee 0)$ where each sub-term is annotated with its state on the right. This formula is accepted by $\mathcal{A}$ because the rules in Δ "rewrite" the input to state q_1, which is a final state.

Definition 2. (Accepting run) *An* accepting run *of an FTA $\mathcal{A} = (Q, Q_f, \Delta)$ is a pair (t, L) where $t = (n_r, V, E)$ 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 (1) $L(n_r) \in Q_f$; (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 $\mathsf{Label}(n)(q_1, \ldots, q_k) \to q$ is a transition in Δ.*

In other words, an accepting run labels each tree node with an automaton state.

Example 2. Let L be the mapping that assigns each node of the tree t in Figure 1 to the state written next to it. Then, (t, L) is an accepting run for Example 1.

4 Constraint Annotated Tree Automata

In this section, we introduce the concept of *Constraint Annotated Tree Automata (CATA)*, which forms the basis of the synthesis algorithm described in the next section.

Definition 3. (CATA) *Let Σ be a finite alphabet and $\mathcal{T}$ be a decidable first-order theory ($\mathcal{T}$ may use symbols from Σ as well as additional symbols). A constraint annotated tree automaton (CATA) over Σ and $\mathcal{T}$ is a tuple $\mathcal{A}_{\mathcal{T}} = (Q, Q_f, \Delta)$ where:*

– *Q is a finite set of states*
– *Q_f is a mapping from states to their acceptance condition, which is a formula in theory $\mathcal{T}$*
– *$\Delta \subseteq \Sigma \times Q^* \times \mathcal{T} \times Q$ is a set of transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow_\varphi q$ where $q, q_1, \ldots, q_n \in Q$ and $\ell \in \Sigma$ and $\varphi \in \mathcal{T}$.*

At a high level, a CATA differs from an FTA in two ways: First, the acceptance condition Q_f is a mapping from each state to a formula φ in theory $\mathcal{T}$. In other words, unlike the standard FTA where Q_f maps each state to a boolean constant, the CATA maps each state to a first-order formula under which that state is accepting. Second, the transitions in a CATA are qualified by formulas in a first-order theory $\mathcal{T}$. In particular, a transition $f(q_1, \ldots, q_n) \rightarrow_\varphi q$ can rewrite $f(q_1, \ldots, q_n)$ to q only if the transition condition φ is satisfied.

Next, we define a *run* of a CATA. Recall that an FTA run consists of a tree and mapping L from nodes of that tree to states in the automaton. For CATAs, we generalize this notion of a run by having two types of mappings: One maps each tree node to a state, and another maps each node to a formula. More formally, we have:

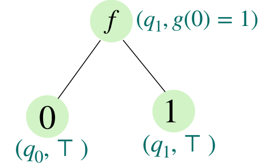


Fig. 2: CATA run on $f(0, 1)$

Definition 4. (CATA run) *Let $\mathcal{A}_\mathcal{T} = (Q, Q_f, \Delta)$ be a CATA over alphabet Σ and theory $\mathcal{T}$. A run of this CATA is a triple $r = (t, L_Q, L_\varphi)$ consisting of:*

1. *A tree $t = (n_r, V, E)$, where each $n \in V$ is labeled by an element of Σ*
2. *A function $L_Q : V \rightarrow Q$ mapping the nodes of t to the states of $\mathcal{A}_\mathcal{T}$*
3. *A function $L_\varphi : V \rightarrow \mathsf{Formulas}(\mathcal{T})$ mapping nodes of t to formulas over theory $\mathcal{T}$ such that if n has label f and children $n_1 \ldots n_k$ then there is a transition*
$$(f(L_Q(n_1), \ldots, L_Q(n_k)) \rightarrow_{L_\varphi(n)} L_Q(n)) \in \Delta$$

In other words, a CATA run not only labels tree nodes with states but also with the conditions under which the corresponding transition is legal. We say that a run (t, m_Q, m_Φ), *ends* at state q if $m_Q(t.\mathsf{Root}) = q$.

Example 3. Consider following CATA $\mathcal{A}_\mathcal{T}$ over the combined theory of uninterpreted functions and integers: $\mathcal{A}_\mathcal{T}$ has states $Q = \{q_0, q_1\}$, $\Sigma_0 = \{0, 1\}$, $\Sigma_2 = \{f\}$, final states $Q_f = \{q_0 \mapsto \bot, q_1 \mapsto g(0) < 1\}$, and the following transitions Δ:

$$1 \rightarrow_\top q_1 \quad 0 \rightarrow_\top q_0 \quad f(q_0, q_0) \rightarrow_{g(0)=0} q_0 \quad f(q_0, q_1) \rightarrow_{g(0)=1} q_1 \; f(q_1, _) \rightarrow_\top q_1$$

Here, $\top, \bot$ denote true and false respectively. Figure 2 shows a run of this CATA, with L_Q, L_φ shown as a pair (q, φ) next to that node.

Definition 5. (Run Assumptions) *Given a run $r = (t, L_Q, L_\varphi)$, the* assumptions *of the run are defined as follows:*

$$\mathsf{Assumptions}(r) = \bigwedge_{n \in \mathsf{Nodes}(t)} L_\varphi(n)$$

For example, the assumptions for the run shown in Figure 2 is just $g(0) = 1$.

Definition 6. (Accepting run) *Given a run $r = (t, L_Q, L_\varphi)$ of CATA $\mathcal{A}_\mathcal{T}$, the* acceptance condition *of the run is the conjunction of the assumptions of r and the formula corresponding to the root node, i.e.,*

$$\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r) = \mathsf{Assumptions}(r) \wedge Q_f(L_Q(\mathsf{Root}(t)))$$

A run r is accepting *if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r)$. We refer to such a model $\mathcal{M}$ as a* witness *for run r.*

Example 4. The run from Example 3 is not accepting because the assumptions made by the run (namely, $g(0) = 1$) contradict the acceptance condition for node q_1, which is $g(0) < 1$.

Recall that an FTA accepts a tree t if there exists a corresponding accepting run for t. We generalize this notion to CATAs as follows:

Definition 7. (Accepted tree) *$\mathcal{A}_\mathcal{T}$ accepts tree t under witness $\mathcal{M}$, denoted $(t, \mathcal{M}) \models \mathcal{A}_\mathcal{T}$, if there is an accepting run $r = (t, L_Q, L_\varphi)$ of $\mathcal{A}_\mathcal{T}$ with witness $\mathcal{M}$.*

Example 5. The tree shown in Figure 2 would be accepting (with the same corresponding run from Figure 2) if we change $Q_f(q_1)$ to $g(0) \geq 1$.

Next, we define the language of a CATA. Recall that the language of an FTA is the set of all trees it accepts. However, since a CATA accepts a tree only under certain conditions, the language of a CATA consists of pairs of trees along with their witnesses. More formally, we have:

$$\mathcal{L}(\mathcal{A}_\mathcal{T}) = \{(t, \mathcal{M}) \mid (t, \mathcal{M}) \models \mathcal{A}_\mathcal{T}\ \}$$

As mentioned earlier, synthesis approaches based on tree automata rely on a *product* operation, $\mathcal{A}^1_\mathcal{T} \times \mathcal{A}^2_\mathcal{T}$, that produces a new automaton $\mathcal{A}_\mathcal{T}$ such that $\mathcal{L}(\mathcal{A}_\mathcal{T}) = \mathcal{L}(\mathcal{A}^1_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}^2_\mathcal{T})$. This operation is defined as follows for CATAs:

Definition 8. (Intersection) *Let $\mathcal{A}^1_\mathcal{T} = (Q_1, Q_{f_1}, \Delta_1)$ and $\mathcal{A}^2_\mathcal{T} = (Q_2, Q_{f_2}, \Delta_2)$ be two CATAs over the same underlying theory $\mathcal{T}$ and alphabet Σ. Then, the* product CATA *$\mathcal{A}^1_\mathcal{T} \times \mathcal{A}^2_\mathcal{T}$ is defined as (Q, Q_f, Δ) where:*

- $Q = Q_1 \times Q_2$
- $Q_f((q_1, q_2)) = Q_{f_1}(q_1) \wedge Q_{f_2}(q_2)$
- *Δ contains the transition $\ell((q_{11}, q_{21}), \ldots, (q_{1n}, q_{2n})) \rightarrow_{\varphi_1 \wedge \varphi_2} (q_1, q_2)$ iff $\ell(q_{11}, \ldots, q_{1n}) \rightarrow_{\varphi_1} q_1 \in \Delta_1$ and $\ell(q_{21}, \ldots, q_{2n}) \rightarrow_{\varphi_2} q_2 \in \Delta_2$*

Example 6. Suppose $\mathcal{A}^1_\mathcal{T}$ contains the transition $q_1 \rightarrow_{g(0) \geq 1} q_2$ and $\mathcal{A}^2_\mathcal{T}$ contains the transition $q_3 \rightarrow_{g(0) \neq 1} q_4$. Then, assuming both CATAs are over the combined theory of integers and uninterpreted functions, the product CATA would contain the transition $(q_1, q_3) \rightarrow_{g(0) > 1} (q_2, q_4)$.

Theorem 1. *Let $\mathcal{A}^1_\mathcal{T} = (Q_1, Q_{f_1}, \Delta_1)$ and $\mathcal{A}^2_\mathcal{T} = (Q_2, Q_{f_2}, \Delta_2)$ be two CATAs over theory $\mathcal{T}$ and alphabet Σ. Then, $\mathcal{L}(\mathcal{A}^1_\mathcal{T} \times \mathcal{A}^2_\mathcal{T}) = \mathcal{L}(\mathcal{A}^1_\mathcal{T}) \cap \mathcal{L}(\mathcal{A}^2_\mathcal{T})$*

4.1 CATA Operations for Synthesis

We now define CATA operations that our synthesis algorithm relies on.

Definition 9. (Accepting Runs of Tree) *Given a CATA $\mathcal{A}_\mathcal{T}$ and tree t, the accepting runs for t, denoted* $\mathsf{Runs}(\mathcal{A}_\mathcal{T}, t)$ *are:*

$$\mathsf{Runs}(\mathcal{A}_\mathcal{T}, t) = \{r = (t, L_Q, L_\varphi) \mid \mathsf{SAT}(\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r))\}$$

In other words, the accepting runs of $\mathcal{A}_\mathcal{T}$ on tree t are those runs whose acceptance conditions are logically satisfiable. We can similarly define accepting runs for a state as all accepting runs that end in that state:

Definition 10. (Accepting Runs of State) *Given a CATA $\mathcal{A}_\mathcal{T}$ and state q, the accepting runs for q, denoted* $\mathsf{Runs}(\mathcal{A}_\mathcal{T}, q)$*, are:*

$$\mathsf{Runs}(\mathcal{A}_\mathcal{T}, q) = \{r = (t, L_Q, L_\varphi) \mid r \in \mathsf{Runs}(\mathcal{A}_\mathcal{T}, t) \wedge L_Q(\mathsf{Root}(t)) = q\}$$

Given a state q or tree t, we often need to compute the acceptance condition for that tree/state, which we define as follows:

Definition 11. (Acceptance Condition) *Given a CATA $\mathcal{A}_\mathcal{T}$ and state or tree x, the acceptance condition of x, denoted* $\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, x)$ *is:*

$$\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, x) = \bigvee_{r \in \mathsf{Runs}(\mathcal{A}_\mathcal{T}, x)} \mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, r)$$

Example 7. Consider $\mathcal{A}_\mathcal{T}$ defined in Example 3, and let t_1 be the tree in Figure 2. We have $\mathsf{Runs}(\mathcal{A}_\mathcal{T}, t_1) = \varnothing$, and $\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T}, q_1) = g(0) < 1$.

Finally, the acceptance condition for the CATA, $\mathsf{AcceptCond}(\mathcal{A}_\mathcal{T})$, is the disjunction of acceptance conditions over all states, and a *minimum accepted tree*, denoted $\mathsf{MinTree}(\mathcal{A}_\mathcal{T})$ is a minimum size tree accepted by the CATA.

5 Synthesis Algorithm

In this section, we first define our synthesis problem more precisely (Section 5.1) and then present the basic synthesis technique (Section 5.2). However, since the basic algorithm ends up requiring too many CATA intersections, it does not lend itself to a practical implementation. In Section 5.3, we show how to construct the CATA in a goal-directed way to minimize the number of CATA intersections.

5.1 Problem Statement

Definition 12. (Relational spec) *Let $\mathcal{F} = \{f_1, \ldots, f_n\}$ be a set of function symbols. A* relational specification *over $\mathcal{F}$ is a formula of the form $\forall \overline{x}.\Phi(\overline{x})$ where Φ is a quantifier-free formula over some theory $\mathcal{T}$ and the only function symbols in Φ belong either in $\mathcal{F}$ or to the signature of $\mathcal{T}$.*

$$
\begin{array}{lllllll}
P & ::= & \texttt{f}(x)\ \texttt{=}\ e\ \texttt{;}\ P \\
e & ::= & x & \mid e_1\ e_2 & \mid \texttt{()} & \mid \texttt{fst}\ e & \mid \texttt{snd}\ e \\
 & \mid & (e_1,\ e_2) & \mid \texttt{inl}\ e & \mid \texttt{inr}\ e & \mid \texttt{unl}\ e & \mid \texttt{unr}\ e \\
 & \mid & \multicolumn{5}{l}{\texttt{switch}\ e_3\ \texttt{on}\ \texttt{inl}\ _ \rightarrow e_1\ \texttt{,}\ \texttt{inr}\ _ \rightarrow e_2} \\
v & ::= & \texttt{()} & \mid (v_1,\ v_2) & \mid \texttt{inl}\ v & \mid \texttt{inr}\ v
\end{array}
$$

Fig. 3: A functional ML-like language. Programs are comprised of a list of mutually recursive function definitions.

$$\frac{P \vdash e \Downarrow v; \varphi_1 \qquad f(x) = e' \in P \qquad P \vdash e'[v/x] \Downarrow v'; \varphi_2}{f\ e \Downarrow v'; \varphi_1 \land \varphi_2 \land f(v) = v'}$$

$$\frac{}{P \vdash \texttt{()} \Downarrow \texttt{()}; \top} \qquad \frac{P \vdash e_1 \Downarrow v_1; \varphi_1 \qquad P \vdash e_2 \Downarrow v_2; \varphi_2}{P \vdash (e_1, e_2) \Downarrow (v_1, v_2); \varphi_1 \land \varphi_2} \qquad \frac{P \vdash e \Downarrow (v_1, v_2); \varphi}{P \vdash \texttt{fst}\ e \Downarrow v_1; \varphi}$$

$$\frac{P \vdash e \Downarrow (v_1, v_2); \varphi}{P \vdash \texttt{snd}\ e \Downarrow v_2; \varphi}$$

$$\frac{P \vdash e \Downarrow v; \varphi}{P \vdash \texttt{inl}\ e \Downarrow \texttt{inl}\ v; \varphi} \qquad \frac{P \vdash e \Downarrow v; \varphi}{P \vdash \texttt{inr}\ e \Downarrow \texttt{inr}\ v; \varphi} \qquad \frac{P \vdash e \Downarrow \texttt{inl}\ v}{P \vdash \texttt{unl}\ e \Downarrow v; \varphi}$$

$$\frac{P \vdash e \Downarrow \texttt{inr}\ v; \varphi}{P \vdash \texttt{unr}\ e \Downarrow v; \varphi} \qquad \frac{P \vdash e_3 \Downarrow \texttt{inl}\ v_3; \varphi_1 \qquad P \vdash e_1 \Downarrow v_1; \varphi_2}{P \vdash \texttt{switch}\ e_3\ \texttt{on}\ \texttt{inl}\ _ \rightarrow e_1\ \texttt{inr}\ _ \rightarrow e_2 \Downarrow v_1; \varphi_1 \land \varphi_2}$$

$$\frac{P \vdash e_3 \Downarrow \texttt{inr}\ v_3; \varphi_1 \qquad P \vdash e_2 \Downarrow v_2; \varphi_2}{P \vdash \texttt{switch}\ e_3\ \texttt{on}\ \texttt{inl}\ _ \rightarrow e_1\ \texttt{inr}\ _ \rightarrow e_2 \Downarrow v_2; \varphi_1 \land \varphi_2}$$

Fig. 4: Program Semantics. The symbols e range over expressions, v range over values, and φ range over formulas in the the theory of uninterpreted functions.

Relational specifications allow jointly constraining the behavior of multiple functions to be synthesized. For instance, examples of relational specifications include $\forall x. f(g(x)) = x$ (i.e., f and g are inverses) or $\forall x, y. f(x, y) = g(y, x)$.

In this paper, we consider the problem of synthesizing programs in an ML-like functional programming language with sums, products, and mutual recursion. Figure 3 shows the core subset of this programming language. A program in this language consists of one or more function definitions, and the body of each function is an expression e, which includes function applications, constructors (`inl`,`inr` for sums and (e_1, e_2) for products), destructors (`unl`,`unr` for sums, `fst`,`snd` for products) and `switch` statements for pattern matching. Figure 4 presents the semantics of this language using the notation $P \vdash e \Downarrow v; \varphi$, meaning that, under the function definitions given by P, expression e evaluates to value v and φ is a formula that tracks the results of procedure calls made by e.[3]

Given a program P in this language defining functions $\mathcal{F}'$ and a relational specification ψ over functions $\mathcal{F} \subseteq \mathcal{F}'$, we write $P \models \psi$ if the implementation of

[3] These *instrumented semantics* for recording results of function calls will be useful for CATA construction in Section 5.

P satisfies specification ψ. Since the focus of this paper is not verification, we assume access to an oracle for checking $P \models \psi$. Given n programs $P_1, \ldots, P_n$ implementing different functions, we also use the notation $(P_1, \ldots, P_n) \models \psi$ to denote that these programs collectively satisfy specification ψ.

Definition 13. (Solution to synthesis problem) *Let ψ be a relational specification over functions $\mathcal{F} = \{f_1, \ldots, f_n\}$. A solution to this synthesis problem is a mapping from each $f_i \in \mathcal{F}$ to a program such P_i such that $(P_1, \ldots, P_n) \models \psi$.*

Since our top-level approach is based on counterexample-guided inductive synthesis (CEGIS) [2], it suffices to have a synthesis procedure that can only deal with *ground relational specifications*. In particular, a ground relational specification over $\mathcal{F}$ cannot contain any variables, either free or bound, besides those in $\mathcal{F}$. In the remainder of this section, we therefore only consider ground specifications and assume that quantifiers are handled using the standard CEGIS framework.

Assumptions. Our synthesis algorithm makes a few important, but realistic assumptions, that we rely on in the remainder of this section. First, we assume that there is a pre-defined partial order relation $\preceq$ between constants in the underlying language (e.g, $1 \prec 3, [1,2] \prec [1,2,3]$ etc). This partial ordering must be well founded and must not have infinite fan-out to ensure termination. Second, we assume that, when function f is called on some input x, other calls that f makes can only involve values satisfying $y \prec x$. This common assumption [1, 24, 33, 35] is required to ensure that recursive calls are well-founded. Finally, to further simplify presentation, we assume that the language admits a finite number of constants; however, our implementation does not make this assumption (see Section 6).

5.2 Basic Synthesis Algorithm

In this section, we describe our CATA-based synthesis procedure. While this algorithm exposes how CATAs are used for synthesis, it does not lend itself to a practical implementation due to its eager nature. We first present the basic algorithm and then explain how to make it more goal-directed in the next section.

Our basic synthesis procedure is summarized in Algorithm 1 and takes two inputs, the set $\mathcal{F}$ of functions to synthesize and a ground relational specification over $\mathcal{F}$. The algorithm computes a solution for each $f \in \mathcal{F}$ in three steps:

- First, for each possible input c of $f \in \mathcal{F}$, the algorithm builds a CATA $\Pi(f, c)$ that encodes how different implementations of f can behave on input c (lines 2–5). In particular, for a possible output c', $\mathsf{AcceptCond}(\Pi(f, c), c')$ gives the conditions under which f can produce c' on input c.
- Next, in lines 6–11, the algorithm builds a CATA, $\Omega(f)$, encoding all possible input-output behaviors of different implementations of f. This is done by using the CATA product operation defined in Section 4. Lines 6–11 also strengthen the initial specification ψ to a stronger condition ϕ by taking into account the acceptance condition of the constructed CATAs.

input: Relational ground specification ψ and set of functions $\mathcal{F}$ to synthesize
output: Solution Ψ mapping each function to its implementation

1: **procedure** Synthesize($\psi, \mathcal{F}$)
▷ Create initial CATAs for each possible input for each function
2: $\Pi \leftarrow \varnothing$ ▷ Mapping from each function and constant to corresponding CATA
3: **for each** $c \in \mathcal{C}$ **do**
4: **for each** $f \in \mathcal{F}$ **do**
5: $\Pi(f, c) \leftarrow$ CreateCATA(f, c, ψ)
▷ Obtain CATA per function and strengthen specification
6: $\phi \leftarrow \psi$ ▷ Initialization for strengthened spec
7: $\Omega \leftarrow \varnothing$ ▷ Mapping from each function to its CATA
8: **for each** $f \in \mathcal{F}$ **do**
9: $\mathcal{A} \leftarrow \Pi(f, c_1) \times \ldots \times \Pi(f, c_n)$
10: $\phi \leftarrow \phi \wedge \mathsf{AcceptCond}(\mathcal{A})$
11: $\Omega(f) \leftarrow \mathcal{A}$
▷ Synthesize implementation of each function
12: $\Psi \leftarrow \varnothing$ ▷ Solution mapping from each function to its implementation
13: **for each** $f \in \mathcal{F}$ **do**
14: $\mathcal{A} \leftarrow \mathsf{StrengthenSpec}(\Omega(f), \phi)$ ▷ Update acceptance condition
15: $P \leftarrow \mathsf{MinTree}(\mathcal{A})$
16: $\phi \leftarrow \phi \wedge \mathsf{AcceptCond}(\mathcal{A}, P)$
17: $\Psi(f) \leftarrow P$
18: **return** Ψ

Algorithm 1: Basic synthesis procedure. $\mathcal{C}$ denotes the set of constants in the programming language, sorted to be consistent with partial order $\preceq$

– Finally, lines 12–17 of the algorithm use the per-function CATA $\Omega(f)$ and the strengthened specification ϕ to obtain a concrete implementation of f. To that end, the algorithm first strengthens the acceptance condition of each state by conjoining the global specification ϕ; it then obtains a minimum tree P accepted by the resulting automaton $\mathcal{A}$. This tree corresponds to the synthesized implementation for f, and the algorithm moves on to the next function after strengthening the global specification ϕ to be consistent with the acceptance condition for P.

The interesting aspect of this algorithm is that it is guaranteed to find a set of programs that collectively satisfy the relational specification without *any* need for backtracking search. Intuitively, there are three key reasons for this:

1. First, when building the CATA for each (f, c) pair, the CreateCATA procedure (formalized as inference rules in Figure 5) generates constraints under which each transition is valid. In particular, consider the Function Call rule in Figure 5. When adding the transition $g(c) \rightarrow c'$, $\mathsf{AcceptCond}(\Pi(g, c), c')$ gives the exact conditions under which g will return c' on input c, and this is the case even when g is one of the functions being synthesized.

$$\textsc{Init}\quad \frac{}{q_{v_{in}} \in Q \qquad \mathtt{x} \rightarrow q_{v_{in}} \in \Delta} \qquad\qquad \textsc{Final}\quad \frac{q_v \in Q}{Q_f(q_v) = \psi \wedge f(v_{in}) = v}$$

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

$$\textsc{Fst}\quad \frac{q_{v_1,v_2} \in Q}{q_{v_1} \in Q \qquad \mathtt{fst}(q_{v_1,v_2}) \rightarrow q_{v_1} \in \Delta} \qquad\qquad \textsc{Inl}\quad \frac{q_v \in Q}{q_{\mathtt{inl}\ v} \in Q \qquad \mathtt{inl}(v) \rightarrow q_{\mathtt{inl}\ v} \in \Delta}$$

$$\textsc{Uneval}\quad \frac{}{\bot \in Q} \qquad \textsc{Uneval Prod}\quad \frac{\ell \in \Sigma}{\bot \in Q \qquad \ell(\bot, \ldots, \bot) \rightarrow \bot} \qquad \textsc{Switch Left}\quad \frac{q_{\mathtt{inl}\ v_3} \in Q \qquad q_{v_1} \in Q}{\mathtt{switch}(q_{\mathtt{inl}\ v_3}, q_{v_1}, \bot) \rightarrow q_{v_1} \in \Delta}$$

$$\textsc{Function Call}\quad \frac{q_v \in Q \qquad \mathcal{A}_{\mathcal{T}} = \text{CATA}(g, v, \psi) \qquad \begin{matrix} v \prec v_{in} \\ q_{v'} \in \mathsf{States}(\mathcal{A}_{\mathcal{T}}) \end{matrix} \qquad \varphi = \mathsf{AcceptCond}(\mathcal{A}_{\mathcal{T}}, q_{v'})}{q_{v'} \in Q \qquad \mathtt{g}(q_v) \rightarrow_{\varphi} q_{v'} \in \Delta}$$

Fig. 5: Inference rules for $\textsc{Create}\text{CATA}(f, v_{in}, \psi)$. CATA states correspond to constants in the language, and we write q_c to denote the state representing constant c. The state $\bot$ corresponds to the non-evaluated branch of a switch. The rules for Snd, Inr, and Switch Right are omitted for space reasons.

2. Second, the strengthened specification ϕ after lines 6-11 precisely encodes all possible *joint* behaviors of all functions to be synthesized. Thus, a model of ϕ corresponds to input-output behaviors of every $f \in \mathcal{F}$ that are both mutually consistent and that will also satisfy the relational specification. Conceptually, by sampling a model $\mathcal{M}$ of ϕ and plugging $\mathcal{M}$ into the transition and acceptance conditions of the CATAs, we can turn each CATA into an FTA and then obtain the solution by finding programs accepted by each FTA.
3. However, one problem with the above model-sampling approach is that it does not guarantee that the synthesized programs are small (e.g., the sampled model may only have very complex implementations). Thus, lines 12–17 of Algorithm 1 construct the model in a lazy way that guarantees minimality at each step. In particular, rather than obtaining a monolithic model of ϕ, the algorithm considers one function f at a time, strengthens its acceptance condition using ϕ, and then finds a minimum size accepting tree P for f. Since P induces certain assumptions on the other functions (or relies on certain assumptions being held), ϕ is gradually concretized (by strengthening it at line 16). Thus, the third step of the synthesis procedure can be viewed as incremental model construction for the formula ϕ obtained after step 2.

Theorem 2. (Soundness of synthesis) *If* $\textsc{Synthesize}(\psi, \mathcal{F})$ *returns* Ψ *such that* $\Psi(f_i) = P_i$*, then we have* $(P_1, \ldots, P_n) \models \psi$*, where* $|\mathcal{F}| = n$*.*

input: Relational ground specification ψ and set of functions $\mathcal{F}$ to synthesize
output: Solution Ψ mapping each function to its implementation

```
1:  procedure LazySynthesize(ψ, 𝓕)
      ▷ Initialization phase
2:    Ω ← {f ↦ 𝒜_𝒯^⊤ | f ∈ 𝓕}                    ▷ Mapping from functions to CATAs
3:    Λ ← ∅        ▷ Mapping from each function to counterexamples that appear in ψ
      ▷ Iteratively refine CATAs until solution is found
4:    while true do
        ▷ Initialization for this refinement iteration
5:      ϕ ← ψ ∧ ⋀_{f∈𝓕} AcceptCond(Ω(f))              ▷ Current global specification
6:      Ψ ← ∅                             ▷ Mapping from functions to candidate solution
        ▷ Get candidate solution
7:      for each f ∈ 𝓕 do
8:        𝒜 ← StrengthenSpec(Ω(f), ϕ)                 ▷ Update acceptance condition
9:        Ψ(f) ← MinTree(𝒜)
10:       ϕ ← ϕ ∧ AcceptCond(𝒜, Ψ(f))
        ▷ Check if Ψ is a valid solution
11:     θ ← {χ | (c′, χ) ∈ Eval(Ψ(f), c), c ∈ Λ(f), f ∈ 𝓕}
12:     if SAT(ψ ∧ ⋀_i θ_i) then
13:       return Ψ                                               ▷ Ψ is a valid solution
        ▷ Refinement phase
14:     γ ← UnsatCore(ψ ∧ ⋀_i θ_i)
15:     for each f(c) ∈ Terms(γ) do
16:       Ω(f) ← Ω(f) × CreateCATA(f, c, ψ)
```

Algorithm 2: Lazy synthesis. $\mathcal{A}_\mathcal{T}^\top$ from line 2 denotes a CATA that accepts all terms, and Eval at line 11 refers to the instrumented semantics (Figure 4).

Theorem 3. (Completeness of synthesis) *Let ψ be a ground relational specification over functions $\mathcal{F}$. If there exists an implementation P_i for each $f_i \in \mathcal{F}$ such that $(P_1, \ldots, P_n) \models \psi$, then* Synthesize *will return a solution.*

5.3 Lazy Synthesis Algorithm

Despite exposing the core ideas underlying our approach, the synthesis algorithm described in Section 5.2 has two severe shortcomings that make it infeasible in practice: First, it considers all possible inputs, which may be very large or even infinite. Second, it eagerly performs CATA intersection, which is impractical due to the exponential blow-up in CATA size. To address these shortcomings, we now describe a *lazy* version of the previous synthesis algorithm that lends itself to a much more practical implementation.[4]

The lazy synthesis procedure is presented in Algorithm 2. As in the previous algorithm, the synthesis procedure maintains a mapping from each function

[4] We note that the eager algorithm as presented in Section 5.2 times out on all of our experimental benchmarks.

$f \in \mathcal{F}$ to its corresponding CATA $\Omega(f)$. However, since $\Omega(f)$ is constructed lazily, $\mathsf{AcceptCond}(\Omega(f))$ *over-approximates* the possible input-output behaviors of f's implementations rather than characterizing them exactly. Thus, lines 4–16 of Algorithm 2 *iteratively refine* Ω as follows until a valid solution is found:

- It first computes the global specification ϕ (line 5) by conjoining all acceptance conditions of the current CATAs with the initial specification ψ.
- Next, it finds a solution Ψ consistent with each CATA and the global specification ϕ, exactly as done in Phase 3 of Algorithm 1 (lines 7–10).
- Then, it checks whether Ψ is a valid solution (lines 11–13). To do so, it executes the candidate implementations on all relevant inputs and tracks the observed input-output behaviors as a set of constraints θ (line 11). If the conjunction of all of these constraints and ψ is satisfiable, then Ψ is indeed a valid solution and is returned at line 13.
- Otherwise, the synthesis procedure obtains an unsat core γ of the resulting unsatisfiable constraint (line 14). Intuitively, if a term $f(c)$ appears in the unsat core, then the CATA for f does not adequately constrain the outputs of f on input c; hence, we must refine $\Omega(f)$ by constructing the CATA for f on this input. Thus, line 16 of Algorithm 2 lazily refines $\Omega(f)$ by considering inputs that appear in the unsat core rather than considering all inputs eagerly.

Our proposed lazy synthesis algorithm is also both sound and complete. The corresponding theorems and proofs are provided in the appendix in the full version of the paper.

Example 8. Consider the evens/odds example from §2. Initially, Ω maps both evens and odds to $\mathcal{A}_{\mathcal{T}}^{T}$, the automaton that accepts all terms for both evens and odds. Suppose, on line 9, Contata obtains $\mathsf{evens}(xs) = []$ and $\mathsf{odds}(xs) = []$ as the solution. Such a solution fails to pass the check on line 12 because it violates the specification that $\mathsf{evens}([x, y, z]) = [x, z]$.

Then, on line 14, Contata computes the unsat core to be $evens([0, 1, 2]) = []$. It now intersects a new CATA created from $evens([0, 1, 2])$ to the evens CATA, which constrains the output of evens.

In the beginning of the next iteration, Contata updates the global specification with the accept condition of constrained automatas. It then pops another candidate program:

```
evens(l) = match l with
  | Nil -> Nil
  | Cons (h,t) -> odds(t)
```

Thus, $\mathsf{evens}(l)$ relies on $\mathsf{odds}([1, 2]) = [1, 2]$. But the odds automaton is unconstrained, and thus will simply return $[]$. So the unsat core will be $\mathsf{odds}([1, 2]) = []$, and so the automaton for $[1, 2]$ would then be intersected with the current odds automaton. This process will continue until eventually the algorithm is able to find a program that relies on valid assumptions.

Benchmark type	Count	Avg. soln size	Example
Mutual recursion (MR)	7	31.0	Test if input is even or odd
Recursive comparators (RC)	7	64.3	Check equality of int-tuple list
Partial data structures (PDS)	12	33.6	Binary tree removal
Stack Overflow (SO)	4	45.5	Reverse a list twice

Table 1: Statistics about the benchmark set

6 Implementation

We have implemented the propsosed algorithm in a new tool called Contata, which is written in OCaml using Z3 for discharging satisfiability queries. In this section, we briefly discuss some implementation details and optimizations elided in the main technical section.

Incremental search. To simplify technical presentation, earlier sections assume that we can build a CATA representing the space of *all* programs consistent with the specification. However, since this space can be very large (or even infinite), Contata builds CATAs of increasing size. In particular, Contata first builds a CATA of size k, increasing the CATA size to $k+1$ if the algorithm fails to find a solution within that search space.

Optimizations. The implementation of Contata includes many standard type-directed synthesis optimizations. For example, to reduce the number of semantically equivalent programs, Contata only considers function implementations that are in eta-long beta-normal form. Additionally, whenever possible, Contata synthesizes generic functions with type parameters to further reduce the search space.

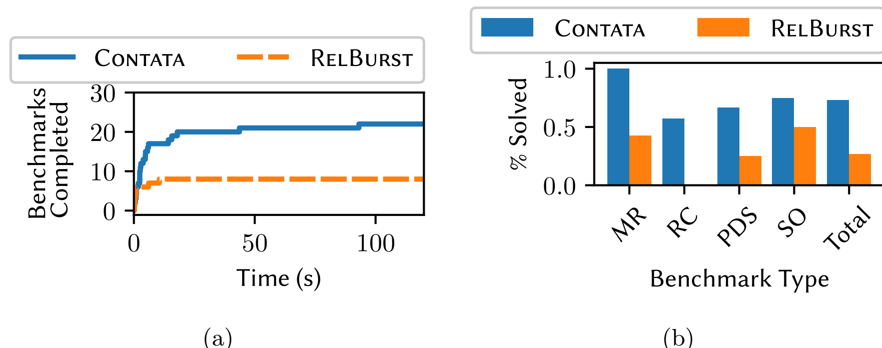
7 Evaluation

In this section, we evaluate Contata through experiments that aim to answer the following research questions:

RQ1. How does Contata compare against prior techniques?

RQ2. What benchmark features impact Contata's performance?

Benchmarks. To answer these questions, we collected a set of 30 benchmarks that exhibit two key characteristics that are relevant to our approach. First, all benchmarks involve recursion or mutual recursion; and, second, the task specification is relational in nature (i.e., relates two different functions to be synthesized or involves a k-safety property [42]). Because the relational specifications we found are often highly unconstrained, we augmented some relational specifications with between 1 and 3 additional input-output examples. The sources of these benchmarks include Stack Overflow posts, functional data structure verification benchmarks [34], and functional programming textbooks [23, 29].



(a) (b)

Fig. 6: (a) The number of benchmarks completed in a given amount of time. (b) The percentage of each class of benchmark solved.

Baseline tool. To the best of our knowledge, there are no existing tools that can solve relational synthesis tasks involving recursion. Thus, to answer RQ1, we implemented another baseline, henceforth referred to as RelBurst, that combines Burst's approach [33] for dealing with recursion with the approach of Relish [51] for handling relational specifications. At a high level, RelBurst first builds FTAs of individual functions using angelic semantics for unknown functions; this introduces many non-deterministic transitions in the FTA. In the second step, RelBurst uses the relational synthesis technique from [51] to construct an automaton representing the specification. Finally, it uses the backtracking search algorithm of [33] to find a set of function implementations that jointly satisfy the relational specification.

Experimental Setup. All of our experiments are conducted on a machine with an Apple M1 Max CPU and 64 GB of physical memory, running the macOS 14.2.1 operating system. For each task, we set the timeout to 2 minutes. In addition to relational specifications for each benchmark, we supply a handful of input-output examples to eliminate the ambiguity of relational constraints.

Results. The results of our evaluation are presented in Figure 6. The plot on the left shows the number of benchmarks solved as we vary the time limit, and the plot on the right compares the percentage of benchmarks solved by Contata against those solved by RelBurst for each class of benchmarks.

Overall, Contata can successfully complete the synthesis task for 22 out of the 30 benchmarks (73%), whereas the baseline completes only 8 (27%). Furthermore, for all completed benchmarks, Contata produces the desired ground truth solution. Additionally, as shown in Figure 6(b), Contata consistently outperforms the baseline across all benchmark categories. Since the key difference between Contata and RelBurst is the use constraint annotated tree automata, these results support our claim that CATAs are useful for reducing

backtracking search when synthesizing recursive programs from relational specifications.

Result for RQ1: Contata solves 2.8× as many benchmarks as a baseline that combines prior techniques for relational synthesis [51] and FTA-based synthesis of recursive procedures [33].

Failure analysis. As shown in 6(b), Contata performs the best on the mutual recursion benchmarks and worst on the recursive comparators. The latter class of benchmarks are particularly difficult; some involve over a hundred AST nodes and multiple recursive calls. As expected, the synthesis algorithm is sensitive to the size of the target program, so the complexity of the ground truth program has a significant impact on running time. However, there are a few benchmarks in the Partial Data Structures category where the size of the synthesized code is relatively small (32 AST nodes) that Contata also times out on. Upon inspection, we noticed that this is due to the "loose" nature of the specification. In such cases, the language of the constructed CATA is quite large, making automaton operations like intersection very expensive. However, this situation can be averted by adding more input-output examples or augmenting the specification with additional constraints.

Result for RQ2: In addition to the complexity of the target program, Contata is sensitive to the precision of the specifications (i.e., performs better with more precise specifications).

8 Related Work

While there is a vast literature on program synthesis, this work is most closely related to techniques that address the synthesis of recursive procedures as well as those that handle relational specifications.

Synthesis of recursive procedures. Research on synthesizing recursive functional programs dates back to the 1970's [26, 43] and has recently become a very active research area [17, 19, 24, 27, 31, 33, 35, 39, 54]. Many of these techniques perform type-directed top-down synthesis from input-output examples [17, 19, 31, 35], whereas Synquid uses refinement types as specifications [39]. Among these approaches, the most related ones are Burst [33], Trio [30], Syrup [54], and SE^{2}GIS [14]. Our technique is directly inspired by Burst and aims to improve upon it by using CATAs to reduce the amount of backtracking search. Trio and Syrup combine deductive reasoning on input-output examples with bottom-up enumeration. In Trio, this is done by generating straight-line programs that are then "folded up" into a recursive program. In contrast, Syrup deduces candidate *recursion traces* to identify possible clusters of valid FTAs to intersect. Both of these approaches rely on input/output deduction and are therefore not easily extensible to the relational synthesis setting that we focus on in this paper.

SE^{2}GIS proves unrealizability of recursion skeletons during synthesis, whereas we use CATAs to rule out incorrect solutions by construction. Additionally, SE^{2}GIS requires a reference implementation and a user-provided recursion skeleton and doesn't consider mutual recursion or relational specifications.

Relational verification and synthesis. This work is also related to a long line of work on reasoning about relational properties [5, 6, 8, 38, 42, 45, 51]. Most techniques in this space address the verification problem and aim to prove a relational property, such as equivalence, between two programs [5, 6, 8, 45]. Some techniques [3, 42] in this space focus on k-safety properties, such as non-interference or associativity, where the goal is to prove that k different executions of the same function do not violate some desired relationship. On the synthesis side, most prior work handles specific classes of tasks, such as program inversion [22] or data type refactoring [10, 36]. To the best of our knowledge, the only synthesis tool that targets a general class of relational properties is Relish [51]. This technique is also based on tree automata and composes FTAs for individual functions in a hierarchical manner by adding non-deterministic transitions between different functions (or different calls to the same function). However, this approach cannot handle recursive or mutually recursive procedures.

Tree automata with constraints. There have been many previous attempts to augment FTAs with constraints. In many of these efforts, e.g., data tree automata [7], finite-memory tree automata [25], and symbolic tree automata [46], the tree alphabet is potentially infinite, and transitions can check constraints over this alphabet. Other work considers finite tree alphabets but imposes global constraints such as the equality and disequality of subtrees [12]. In contrast to these FTA variants, transitions in our proposed CATA model can use symbols from outside the tree alphabet, and the CATA's models are not directly tied to labels of the input tree. To our knowledge, such an automaton model has not been considered in the literature.

9 Conclusion

In this paper, we introduced constraint annotated tree automata (CATA) and developed a program synthesis algorithm based on CATAs. Notably, our proposed algorithm can synthesize recursive and mutually-recursive functions from relational specifications. We also implemented this algorithm in a tool called Contata and showed experimentally that Contata outperforms prior approaches by avoiding backtracking search.

While our approach enables solving synthesis tasks that are out of scope for prior approaches, there remains significant future work in solving relational synthesis tasks involving recursion. In future work, we plan to explore the combination of CATAs with top-down synthesis and ML guidance.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 934–950. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8˙67
2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. IEEE (2013)
3. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. ACM SIGPLAN Notices **52**(6), 362–375 (2017)
4. Barnaby, C., Chen, Q., Samanta, R., Dillig, I.: Imageeye: Batch image processing using program synthesis. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023). https://doi.org/10.1145/3591248, https://doi.org/10.1145/3591248
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: International Symposium on Formal Methods. pp. 200–214. Springer (2011)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. ACM SIGPLAN Notices **39**(1), 14–25 (2004)
7. Björklund, H., Bojańczyk, M.: Bounded depth data trees. In: Automata, Languages and Programming: 34th International Colloquium, ICALP 2007, Wrocław, Poland, July 9-13, 2007. Proceedings 34. pp. 862–874. Springer (2007)
8. Chen, J., Wei, J., Feng, Y., Bastani, O., Dillig, I.: Relational verification using reinforcement learning. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019)
9. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation. pp. 487–502 (2020)
10. Chen, Y., Wang, Y., Goyal, M., Dong, J., Feng, Y., Dillig, I.: Synthesis-powered optimization of smart contracts via data type refactoring. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 560–588 (2022)
11. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. SIGPLAN Not. **35**(9), 268–279 (sep 2000). https://doi.org/10.1145/357766.351266, https://doi.org/10.1145/357766.351266
12. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: Tree automata techniques and applications (2008)
13. De Halleux, J., Tillmann, N.: Parameterized unit testing with pex: (tutorial). In: Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings 2. pp. 171–181. Springer (2008)
14. Farzan, A., Lette, D., Nicolet, V.: Recursion synthesis with unrealizability witnesses. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 244–259. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523726, https://doi.org/10.1145/3519939.3523726
15. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. SIGPLAN Not. **53**(4), 420–435 (jun 2018). https://doi.org/10.1145/3296979.3192382, https://doi.org/10.1145/3296979.3192382

16. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 422–436. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062351, https://doi.org/10.1145/3062341.3062351
17. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (Jun 2015). https://doi.org/10.1145/2813885.2737977, https://doi.org/10.1145/2813885.2737977
18. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes **22**(4), 74–80 (1997)
19. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: A type-theoretic interpretation. SIGPLAN Not. **51**(1), 802–815 (Jan 2016). https://doi.org/10.1145/2914770.2837629, https://doi.org/10.1145/2914770.2837629
20. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 317–330. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926423, https://doi.org/10.1145/1926385.1926423
21. Guo, Z., Cao, D., Tjong, D., Yang, J., Schlesinger, C., Polikarpova, N.: Type-directed program synthesis for restful apis. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 122–136. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523450, https://doi.org/10.1145/3519939.3523450
22. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 376–389 (2017)
23. Hutton, G.: Programming in Haskell. Cambridge University Press, USA, 2nd edn. (2016)
24. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 944–959. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454087, https://doi.org/10.1145/3453483.3454087
25. Kaminski, M., Tan, T.: Tree automata over infinite alphabets. Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday pp. 386–423 (2008)
26. Kitzelmann, E., Schmid, U., Olsson, R., Kaelbling, L.P.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7**(2) (2006)
27. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. p. 407–426. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509555, https://doi.org/10.1145/2509136.2509555

28. Lampropoulos, L., Hicks, M., Pierce, B.C.: Coverage guided, property based testing. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–29 (2019)
29. Lampropoulos, L., Pierce, B.C.: QuickChick: Property-Based Testing in Coq. Software Foundations series, volume 4, Electronic textbook (Aug 2018)
30. Lee, W., Cho, H.: Inductive synthesis of structurally recursive functional programs from non-recursive expressions. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). https://doi.org/10.1145/3571263, https://doi.org/10.1145/3571263
31. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020). https://doi.org/10.1145/3408991, https://doi.org/10.1145/3408991
32. Mariano, B., Chen, Y., Feng, Y., Durrett, G., Dillig, I.: Automated transpilation of imperative to functional code using neural-guided program synthesis. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527315, https://doi.org/10.1145/3527315
33. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up synthesis of recursive functional programs using angelic execution. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498682, https://doi.org/10.1145/3498682
34. Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1–15. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3385967, https://doi.org/10.1145/3385412.3385967
35. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 619–630. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2738007, https://doi.org/10.1145/2737924.2738007
36. Pailoor, S., Wang, Y., Dillig, I.: Semantic code refactoring for abstract data types. Proc. ACM Program. Lang. **8**(POPL) (January 2024). https://doi.org/10.1145/3632870, https://dl.acm.org/doi/10.1145/3632870
37. Pailoor, S., Wang, Y., Wang, X., Dillig, I.: Synthesizing data structure refinements from integrity constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 574–587. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454063, https://doi.org/10.1145/3453483.3454063
38. Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30. pp. 164–182. Springer (2018)
39. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 522–538. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908093, https://doi.org/10.1145/2908080.2908093

40. Samak, M., Kim, D., Rinard, M.C.: Synthesizing replacement classes. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371120, https://doi.org/10.1145/3371120
41. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 404–415. ACM (2006). https://doi.org/10.1145/1168857.1168907, https://doi.org/10.1145/1168857.1168907
42. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. SIGPLAN Not. **51**(6), 57–69 (jun 2016). https://doi.org/10.1145/2980983.2908092, https://doi.org/10.1145/2980983.2908092
43. Summers, P.D.: A methodology for lisp program construction from examples. J. ACM **24**(1), 161–175 (jan 1977). https://doi.org/10.1145/321992.322002, https://doi.org/10.1145/321992.322002
44. Tillmann, N., Schulte, W.: Parameterized unit tests. ACM SIGSOFT Software Engineering Notes **30**(5), 253–262 (2005)
45. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: International Conference on Computer Aided Verification. pp. 742–766. Springer (2021)
46. Veanes, M., Bjørner, N.: Symbolic tree automata. Information Processing Letters **115**(3), 418–424 (2015)
47. Wang, C., Cheung, A., Bodik, R.: Interactive query synthesis from input-output examples. In: Proceedings of the 2017 ACM International Conference on Management of Data. p. 1631–1634. SIGMOD '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3058738, https://doi.org/10.1145/3035918.3058738
48. Wang, C., Feng, Y., Bodik, R., Dillig, I., Cheung, A., Ko, A.J.: Falx: Synthesis-powered visualization authoring. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. CHI '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3411764.3445249, https://doi.org/10.1145/3411764.3445249
49. Wang, X., Dillig, I., Singh, R.: Program synthesis using abstraction refinement. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158151, https://doi.org/10.1145/3158151
50. Wang, X., Dillig, I., Singh, R.: Synthesis of data completion scripts using finite tree automata. Proc. ACM Program. Lang. **1**(OOPSLA) (Oct 2017). https://doi.org/10.1145/3133886, https://doi.org/10.1145/3133886
51. Wang, Y., Wang, X., Dillig, I.: Relational program synthesis. Proc. ACM Program. Lang. **2**(OOPSLA) (Oct 2018). https://doi.org/10.1145/3276525, https://doi.org/10.1145/3276525
52. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: Sqlizer: query synthesis from natural language. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–26 (2017)
53. Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-guided inductive synthesis of recursive functional programs. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023). https://doi.org/10.1145/3591255, https://doi.org/10.1145/3591255
54. Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-guided inductive synthesis of recursive functional programs. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023). https://doi.org/10.1145/3591255, https://doi.org/10.1145/3591255