Relational Program Synthesis

YUEPENG WANG, University of Texas at Austin, USA
XINYU WANG, University of Texas at Austin, USA
ISIL DILLIG, University of Texas at Austin, USA

This paper proposes *relational program synthesis*, a new problem that concerns synthesizing one or more programs that collectively satisfy a *relational specification*. As a dual of relational program verification, relational program synthesis is an important problem that has many practical applications, such as automated program inversion and automatic generation of comparators. However, this relational synthesis problem introduces new challenges over its non-relational counterpart due to the combinatorially larger search space. As a first step towards solving this problem, this paper presents a synthesis technique that combines the counterexample-guided inductive synthesis framework with a novel inductive synthesis algorithm that is based on *relational version space learning*. We have implemented the proposed technique in a framework called Relish, which can be instantiated to different application domains by providing a suitable domain-specific language and the relevant relational specification. We have used the Relish framework to build relational synthesizers to automatically generate string encoders/decoders as well as comparators, and we evaluate our tool on several benchmarks taken from prior work and online forums. Our experimental results show that the proposed technique can solve almost all of these benchmarks and that it significantly outperforms EUSolver, a generic synthesis framework that won the general track of the most recent SyGuS competition.

CCS Concepts: • **Software and its engineering → Programming by example**; **Automatic programming**; • **Theory of computation → Formal languages and automata theory**;

Additional Key Words and Phrases: Relational Program Synthesis, Version Space Learning, Counterexample Guided Inductive Synthesis.

ACM Reference Format:
Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (November 2018), 26 pages. https://doi.org/10.1145/3276525

1 INTRODUCTION

Relational properties describe requirements on the interaction between multiple programs or different runs of the same program. Examples of relational properties include the following:

- *Equivalence:* Are two programs P_1, P_2 observationally equivalent? (i.e., $\forall \vec{x}.\ P_1(\vec{x}) = P_2(\vec{x})$)
- *Inversion:* Given two programs P_1, P_2, are they inverses of each other? (i.e., $\forall x.\ P_2(P_1(x)) = x$)
- *Non-interference:* Given a program P with two types of inputs, namely *low* (public) input $\vec{l}$ and *high* (secret) input $\vec{h}$, does P produce the same output when run on the same low input $\vec{l}$ but two different high inputs $\vec{h_1}, \vec{h_2}$? (i.e., $\forall \vec{l}, \vec{h_1}, \vec{h_2}.\ P(\vec{l}, \vec{h_1}) = P(\vec{l}, \vec{h_2})$)
- *Transitivity:* Given a program P that returns a boolean value, does P obey transitivity? (i.e., $\forall x, y, z.\ P(x, y) \wedge P(y, z) \Rightarrow P(x, z)$)

Authors' addresses: Yuepeng Wang, University of Texas at Austin, USA, ypwang@cs.utexas.edu; Xinyu Wang, University of Texas at Austin, USA, xwang@cs.utexas.edu; Isil Dillig, University of Texas at Austin, USA, isil@cs.utexas.edu.

2475-1421/2018/11-ART155
https://doi.org/10.1145/3276525

Observe that the first two properties listed above relate different programs, while the latter two relate multiple runs of the same program.

Due to their importance in a wide range of application domains, relational properties have received significant attention from the program verification community. For example, prior papers propose novel *program logics* for verifying relational properties [Barthe et al. 2012b; Benton 2004; Chen et al. 2017; Sousa and Dillig 2016; Yang 2007] or transform the relational verification problem to standard safety by constructing so-called *product programs* [Barthe et al. 2011, 2013, 2016].

In this paper, we consider the dual *synthesis* problem of relational verification. That is, given a relational specification Ψ relating runs of n programs $P_1, \ldots, P_n$, our goal is to automatically synthesize n programs that satisfy Ψ. This *relational synthesis* problem has a broad range of practical applications. For example, we can use relational synthesis to automatically generate comparators that provably satisfy certain correctness requirements such as transitivity and anti-symmetry. As another example, we can use relational synthesis to solve the *program inversion* problem where the goal is to generate a program P' that is an inverse of another program P [Hu and D'Antoni 2017; Srivastava et al. 2011]. Furthermore, since many automated repair techniques rely on program synthesis [Jobstmann et al. 2005; Nguyen et al. 2013], we believe that relational synthesis could also be useful for repairing programs that violate a relational property like non-interference.

Solving the relational synthesis problem introduces new challenges over its non-relational counterpart due to the combinatorially larger search space. In particular, a naive algorithm that simply enumerates combinations of programs and then checks their correctness with respect to the relational specification is unlikely to scale. Instead, we need to design novel *relational* synthesis algorithms to efficiently search for *tuples of programs* that collectively satisfy the given specification.

We solve this challenge by introducing a novel synthesis algorithm that learns *relational version spaces (RVS)*, a generalization of the notion of *version space* utilized in prior work [Gulwani 2011; Lau et al. 2003; Wang et al. 2017]. Similar to other synthesis algorithms based on counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama 2008], our method also alternates between inductive synthesis and verification; but the counterexamples returned by the verifier are relational in nature. Specifically, given a set of *relational counterexamples*, such as $f(1) = g(1)$ or $f(g(1), g(2)) = f(3)$, our inductive synthesizer compactly represents *tuples* of programs and efficiently searches for their implementations that satisfy all relational counterexamples.

In more detail, our relational version space learning algorithm is based on the novel concept of *hierarchical finite tree automata (HFTA)*. Specifically, an HFTA is a hierarchical collection of finite tree automata (FTAs), where each individual FTA represents possible implementations of the different functions to be synthesized. Because relational counterexamples can refer to compositions of functions (e.g., $f(g(1), g(2))$), the HFTA representation allows us to compose different FTAs according to the hierarchical structure of subterms in the relational counterexamples. Furthermore, our method constructs the HFTA in such a way that tuples of programs that do not satisfy the examples are rejected and therefore excluded from the search space. Thus, the HFTA representation allows us to compactly represent those programs that are consistent with the relational examples.

We have implemented the proposed relational synthesis algorithm in a tool called Relish [1] and evaluate it in the context of two different relational properties. First, we use Relish to automatically synthesize encoder-decoder pairs that are provably inverses of each other. Second, we use Relish to automatically generate comparators, which must satisfy three different relational properties, namely, anti-symmetry, transitivity, and totality. Our evaluation shows that Relish can efficiently solve interesting benchmarks taken from previous literature and online forums. Our evaluation

[1] Relish stands for RELatIonal SyntHesis.

Property	Relational specification
Equivalence	$\forall \vec{x}.\ f_1(\vec{x}) = f_2(\vec{x})$
Commutativity	$\forall x.\ f_1(f_2(x)) = f_2(f_1(x))$
Distributivity	$\forall x, y, z.\ f_2(f_1(x, y), z) = f_1(f_2(x, z), f_2(y, z))$
Associativity	$\forall x, y, z.\ f(f(x, y), z) = f(x, f(y, z))$
Anti-symmetry	$\forall x, y.\ (f(x, y) = \mathit{true} \wedge x \neq y) \Rightarrow f(y, x) = \mathit{false}$

Fig. 1. Examples of relational specifications for five relational properties.

also shows that Relish significantly outperforms EUSolver, a general-purpose synthesis tool that won the General Track of the most recent SyGuS competition for syntax-guided synthesis.

To summarize, this paper makes the following key contributions:

- We introduce the *relational synthesis problem* and take a first step towards solving it.
- We describe a *relational version space* learning algorithm based on the concept of *hierarchical finite tree automata* (HFTA).
- We show how to construct HFTAs from relational examples expressed as ground formulas and describe an algorithm for finding the desired accepting runs.
- We experimentally evaluate our approach in two different application domains and demonstrate the advantages of our relational version space learning approach over a state-of-the-art synthesizer based on enumerative search.

2 OVERVIEW

In this section, we define the *relational synthesis* problem and give a few motivating examples.

2.1 Problem Statement

The input to our synthesis algorithm is a *relational specification* defined as follows:

*Definition 2.1 (**Relational specification**).* A relational specification with functions $f_1, \ldots, f_n$ is a first-order sentence $\Psi = \forall \vec{x}.\ \phi(\vec{x})$, where $\phi(\vec{x})$ is a quantifier-free formula with uninterpreted functions $f_1, \ldots, f_n$.

Fig. 1 shows some familiar relational properties like distributivity and associativity as well as their corresponding specifications. Even though we refer to Definition 2.1 as a *relational specification*, observe that it allows us to express combinations of both relational and non-relational properties. For instance, the specification $\forall x.\ (f(x) \geq 0 \wedge f(x) + g(x) = 0)$ imposes both a non-relational property on f (namely, that all of its outputs must be non-negative) as well as the relational property that the outputs of f and g must always add up to zero.

*Definition 2.2 (**Relational program synthesis**).* Given a set of n function symbols $\mathcal{F} = \{f_1, \ldots, f_n\}$, their corresponding domain-specific languages $\{L_1, \ldots, L_n\}$, and a relational specification Ψ, the relational program synthesis problem is to find an interpretation $\mathcal{I}$ for $\mathcal{F}$ such that:

(1) For every function symbol $f_i \in \mathcal{F}$, $\mathcal{I}(f_i)$ is a program in f_i's DSL L_i (i.e., $\mathcal{I}(f_i) \in L_i$).
(2) The interpretation $\mathcal{I}$ satisfies the relational specification Ψ (i.e., $\mathcal{I}(f_1), \ldots, \mathcal{I}(f_n) \models \Psi$).

2.2 Motivating Examples

We now illustrate the practical relevance of the relational program synthesis problem through several real-world programming scenarios.

*Example 2.3 (**String encoders and decoders**).* Consider a programmer who needs to implement a Base64 encoder `encode(x)` and its corresponding decoder `decoder(x)` for any Unicode string `x`.

For example, according to Wikipedia [2], the encoder should transform the string "Man" into "TWFu", "Ma" into "TWE=", and "M" to "TQ==". In addition, applying the decoder to the encoded string should yield the original string. Implementing this encoder/decoder pair is a relational synthesis problem in which the relational specification is the following:

$$\begin{array}{ll} & \texttt{encode}(\text{“Man”}) = \text{“TWFu”} \wedge \texttt{encode}(\text{“Ma”}) = \text{“TWE=”} \wedge \texttt{encode}(\text{“M”}) = \text{“TQ==”} \quad (\textit{input-output examples}) \\ \wedge & \forall x.\ \texttt{decode}(\texttt{encode}(x)) = x \quad (\textit{inversion}) \end{array}$$

Here, the first part (i.e., the first line) of the specification gives three input-output examples for encode, and the second part states that decode must be the inverse of encode. Relish can automatically synthesize the correct Base64 encoder and decoder from this specification using a DSL targeted for this domain (see Section 6.2).

*Example 2.4 (**Comparators**).* Consider a programmer who needs to implement a comparator for sorting an array of integers according to the number of occurrences of the number 5 [3]. Specifically, `compare(x,y)` should return `-1` (resp. `1`) if `x` (resp. `y`) contains less 5's than `y` (resp. `x`), and ties should be broken based on the actual values of the integers. For instance, sorting the array ["24","15","55","101","555"] using this comparator should yield array ["24","101","15","55","555"]. Furthermore, since the comparator must define a total order, its implementation should satisfy reflexivity, anti-symmetry, transitivity, and totality. The problem of generating a suitable `compare` method is again a relational synthesis problem and can be defined using the following specification:

$$\begin{array}{ll} & \texttt{compare}(\text{“24”}, \text{“15”}) = -1 \wedge \texttt{compare}(\text{“101”}, \text{“24”}) = 1 \wedge \ldots \quad (\textit{input-output examples}) \\ \wedge & \forall x.\ \texttt{compare}(x, x) = 0 \quad (\textit{reflexivity}) \\ \wedge & \forall x, y.\ \textit{sgn}(\texttt{compare}(x, y)) = -\textit{sgn}(\texttt{compare}(y, x)) \quad (\textit{anti-symmetry}) \\ \wedge & \forall x, y, z.\ \texttt{compare}(x, y) > 0 \wedge \texttt{compare}(y, z) > 0 \Rightarrow \texttt{compare}(x, z) > 0 \quad (\textit{transitivity}) \\ \wedge & \forall x, y, z.\ \texttt{compare}(x, y) = 0 \Rightarrow \textit{sgn}(\texttt{compare}(x, z)) = \textit{sgn}(\texttt{compare}(y, z)) \quad (\textit{totality}) \end{array}$$

A solution to this problem is given by the following implementation of `compare`:

```
let a = intCompare (countChar x '5') (countChar y '5') in
    if a != 0 then a else intCompare (toInt x) (toInt y)
```

where `countChar` function returns the number of occurrences of a character in the input string, and `intCompare` is the standard comparator on integers.

*Example 2.5 (**Equals and hashcode**).* A common programming task is to implement `equals` and `hashcode` methods for a given class. These functions are closely related because `equals(x,y)=true` implies that the hash codes of `x` and `y` must be the same. Relational synthesis can be used to simultaneously generate implementations of `equals` and `hashcode`. For example, consider an `ExperimentResults` class that internally maintains an array of numbers where negative integers indicate an anomaly (i.e., failed experiment) and should be ignored when comparing the results of two experiments. For instance, the results `[23.5,-1,34.7]` and `[23.5,34.7]` should be equal whereas `[23.5,34.7]` and `[34.7,23.5]` should not. The programmer can use a relational synthesizer to generate `equals` and `hashcode` implementations by providing the following specification:

$$\begin{array}{ll} & \texttt{equals}([23.5,-1,34.7],[23.5,34.7]) = \texttt{true} \quad (\textit{input-output examples}) \\ \wedge & \texttt{equals}([23.5,34.7],[34.7,23.5]) = \texttt{false} \wedge \ldots \\ \wedge & \forall x, y.\ \texttt{equals}(x, y) \Rightarrow (\texttt{hashcode}(x) = \texttt{hashcode}(y)) \quad (\textit{equals-hashcode}) \end{array}$$

A possible solution consists of the following pair of implementations of `equals` and `hashcode`:

```
equals(x,y) : (filter (>= 0) x) == (filter (>= 0) y)
hashcode(x) : foldl (\u.\v. 31 * (u + v)) 0 (filter (>= 0) x)
```

[2] https://en.wikipedia.org/wiki/Base64
[3] https://stackoverflow.com/questions/19231727/sort-array-based-on-number-of-character-occurrences

*Example 2.6 (**Reducers**).* MapReduce is a software framework for writing applications that process large datasets in parallel. Users of this framework need to implement both a *mapper* that maps input key/value pairs to intermediate ones as well as a *reducer* which transforms intermediate values that share a key to a smaller set of values. An important requirement in the MapReduce framework is that the `reduce` function must be associative. Now, consider the task of using the MapReduce framework to sum up sensor measurements obtained from an experiment. In particular, each sensor value is either a real number or `None`, and the final result should be `None` if *any* sensor value is `None`. In order to implement this functionality, the user needs to write a reducer that takes sensor values `x`, `y` and returns their sum if neither of them is `None`, or returns `None` otherwise. We can express this problem using the following relational specification:

$$\begin{array}{lll} & \texttt{reduce(1, 2) = 3} \wedge \texttt{reduce(1, None) = None} & (\textit{input-output examples}) \\ \wedge & \forall x, y, z.\ \texttt{reduce(reduce}(x, y), z) = \texttt{reduce}(x, \texttt{reduce}(y, z)) & (\textit{associativity}) \end{array}$$

The first part of the specification gives input-output examples to illustrate the desired functionality, and the latter part expresses the associativity requirement on `reduce`. A possible solution to this relational synthesis problem is given by the following implementation:

```
reduce(x,y) : if x == None then None
              else if y == None then None
              else x + y
```

3 PRELIMINARIES

Since the rest of this paper requires knowledge of finite tree automata and their use in program synthesis, we first briefly review some background material.

3.1 Finite Tree Automata

A tree automaton is a type of state machine that recognizes trees rather than strings. More formally, a (bottom-up) *finite tree automaton* (FTA) is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, where

- Q is a finite set of states.
- Σ is an alphabet.
- $Q_f \subseteq Q$ is a set of final states.
- $\Delta \subseteq Q^* \times \Sigma \times Q$ is a set of transitions (or rewrite rules).

Intuitively, a tree automaton $\mathcal{A}$ recognizes a term (i.e., tree) t if we can rewrite t into a final state $q \in Q_f$ using the rewrite rules given by Δ.

More formally, suppose we have a tree $t = (V, E, v_r)$ where V is a set of nodes labeled with $\sigma \in \Sigma$, $E \subseteq V \times V$ is a set of edges, and $v_r \in V$ is the root node. A *run* of an FTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ on t is a mapping $\pi : V \to Q$ compatible with Δ (i.e., given node v with label σ and children $v_1, \ldots, v_n$ in the tree, the run π can only map $v, v_1, \ldots, v_n$ to $q, q_1, \ldots, q_n$ if there is a transition $\sigma(q_1, \ldots, q_n) \to q$ in Δ). Run π is said to be *accepting* if it maps the root node of t to a final state, and a tree t is *accepted by* an FTA $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The *language* $\mathcal{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all those trees that are accepted by $\mathcal{A}$.

Example 3.1. Consider a finite tree automaton $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ with states $Q = \{q_0, q_1\}$, alphabet $\Sigma = \{0, 1, \neg, \wedge\}$, final states $Q_f = \{q_1\}$, and transitions Δ:

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

Intuitively, the states of this FTA correspond to boolean constants (i.e., q_0, q_1 represent false and true respectively), and the transitions define the semantics of the boolean connectives $\neg$ and $\wedge$. Since the final state is q_1, the FTA accepts all boolean formulas (containing only $\wedge$ and $\neg$) that

$$\frac{e_{in} = (e_1, \ldots, e_n)}{G, e \;\vdash\; q^{e_i}_{x_i} \in Q,\;\; x_i \to q^{e_i}_{x_i} \in \Delta} \text{ (Input)} \qquad \frac{}{G, e \;\vdash\; q^{e_{out}}_{s_0} \in Q_f} \text{ (Output)}$$

$$\frac{(s \to \sigma(s_1, \ldots, s_n)) \in P \quad G, e \vdash q^{c_1}_{s_1} \in Q, \ldots, q^{c_n}_{s_n} \in Q \quad c = [\![\sigma(c_1, \ldots, c_n)]\!]}{G = (T, N, P, s_0), e \;\vdash\; q^{c}_{s} \in Q,\;\; \sigma(q^{c_1}_{s_1}, \ldots, q^{c_n}_{s_n}) \to q^{c}_{s} \in \Delta} \text{ (Prod)}$$

Fig. 3. Rules for constructing FTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ given example $e = (e_{in}, e_{out})$ and grammar $G = (T, N, P, s_0)$. The alphabet Σ of the FTA is exactly the terminals T of G.

evaluate to *true*. For example, $1 \wedge \neg 0$ is accepted by $\mathcal{A}$, and the corresponding tree and its accepting run are shown in Fig. 2.

3.2 Example-based Synthesis using FTAs

Since our relational synthesis algorithm leverages prior work on example-based synthesis using FTAs [Wang et al. 2017, 2018a], we briefly review how FTAs can be used for program synthesis.

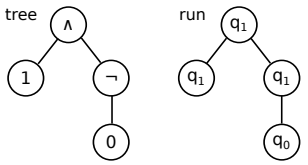

Fig. 2. Tree for $1 \wedge \neg 0$ and its accepting run.

At a high-level, the idea is to build an FTA that accepts *exactly* the ASTs of those DSL programs that are consistent with the given input-output examples. The states of this FTA correspond to concrete values, and the transitions are constructed using the DSL's semantics. In particular, the FTA states correspond to output values of DSL programs on the input examples, and the final states of the FTA are determined by the given output examples. Once this FTA is constructed, the synthesis task boils down to finding an accepting run of the FTA. A key advantage of using FTAs for synthesis is to enable search space reduction by allowing sharing between programs that have the same input-output behavior.

In more detail, suppose we are given a set of input-output examples $\vec{e}$ and a context-free grammar G describing the target domain-specific language. We assume that G is of the form (T, N, P, s_0) where T and N are the terminal and non-terminal symbols respectively, P is a set of productions of the form $s \to \sigma(s_1, \cdots, s_n)$, and $s_0 \in N$ is the topmost non-terminal (start symbol) in G. We also assume that the program to be synthesized takes arguments $x_1, \ldots, x_n$.

Fig. 3 reviews the construction rules for a single example $e = (e_{in}, e_{out})$ [Wang et al. 2018a]. In particular, the Input rule creates n initial states $q^{e_1}_{x_1}, \ldots, q^{e_n}_{x_n}$ for input example $e_{in} = (e_1, \ldots, e_n)$. We then iteratively use the Prod rule to generate new states and transitions using the productions in grammar G and the concrete semantics of the DSL constructs associated with the production. Finally, according to the Output rule, the only final state is $q^{c}_{s_0}$ where s_0 is the start symbol and c is the output example e_{out}. Observe that we can build an FTA for a *set* of input-output examples by constructing the FTA for each example individually and then taking their intersection using standard techniques [Comon 1997].

Remark. Since the rules in Fig. 3 may generate an infinite set of states and transitions, the FTA is constructed by applying the Prod rule a finite number of times. The number of applications of the Prod rule is determined by a bound on the AST depth of the synthesized program.

Example 3.2. Consider a very simple DSL specified by the following CFG, where the notation $s \to s'$ is short-hand for $s \to id(s')$ for a unary identity function id:

$$e \;\to\; x_1 \mid x_2 \mid e + e$$

Suppose we want to find a program (of size at most 3) that is consistent with the input-output example $(1, 3) \rightarrow 4$. Using the rules from Fig. 3, we can construct $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ with states $Q = \{q^1_{x_1}, q^3_{x_2}, q^1_e, q^2_e, q^3_e, q^4_e, q^6_e\}$, alphabet $\Sigma = \{x_1, x_2, id, +\}$, final states $Q_f = \{q^4_e\}$, and transitions Δ:

$$\begin{array}{llll} x_1 \rightarrow q^1_{x_1} & x_2 \rightarrow q^3_{x_2} & id(q^1_{x_1}) \rightarrow q^1_e & id(q^3_{x_2}) \rightarrow q^3_e \\ +(q^1_e, q^1_e) \rightarrow q^2_e & +(q^1_e, q^3_e) \rightarrow q^4_e & +(q^3_e, q^1_e) \rightarrow q^4_e & +(q^3_e, q^3_e) \rightarrow q^6_e \end{array}$$

For example, the FTA contains the transition $+(q^3_e, q^1_e) \rightarrow q^4_e$ because the grammar contains a production $e \rightarrow e + e$ and we have $3 + 1 = 4$. Since it is sometimes convenient to visualize FTAs as hypergraphs, Fig. 4 shows a hypergraph representation of this FTA, using circles to indicate states, double circles to indicate final states, and labeled (hyper-)edges to represent transitions. The transitions of nullary alphabet symbols (i.e. $x_1 \rightarrow q^1_{x_1}, x_2 \rightarrow q^3_{x_2}$) are omitted in the hypergraph for brevity. Note that the only two programs that are accepted by this FTA are $x_1 + x_2$ and $x_2 + x_1$.

4 HIERARCHICAL FINITE TREE AUTOMATA

As mentioned in Section 1, our relational synthesis technique uses a version space learning algorithm that is based on the novel concept of *hierarchical finite tree automata (HFTA)*. Thus, we first introduce HFTAs in this section before describing our relational synthesis algorithm.

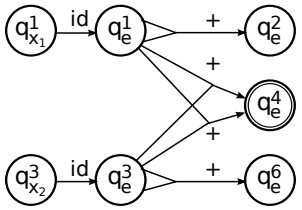

Fig. 4. FTA for input-output $(1, 3) \rightarrow 4$.

A hierarchical finite tree automaton (HFTA) is a tree in which each node is annotated with a finite tree automaton (FTA). More formally, an HFTA is a tuple $\mathcal{H} = (V, \Omega, v_r, \Lambda)$ where

- V is a finite set of nodes.
- Ω is a mapping that maps each node $v \in V$ to a finite tree automaton.
- $v_r \in V$ is the root node.
- $\Lambda \subseteq States(Range(\Omega)) \times States(Range(\Omega))$ is a set of inter-FTA transitions.

Intuitively, an HFTA is a tree-structured (or hierarchical) collection of FTAs, where Λ corresponds to the edges of the tree and specifies how to transition from a state of a child FTA to a state of its parent. For instance, the left-hand side of Fig. 5 shows an HFTA, where the inter-FTA transitions Λ (indicated by dashed lines) correspond to the edges (v_3, v_1) and (v_3, v_2) in the tree.

Just as FTAs accept trees, HFTAs accept *hierarchical trees*. Intuitively, as depicted in the right-hand side of Fig. 5, a hierarchical tree organizes a collection of trees in a tree-structured (i.e., hierarchical) manner. More formally, a hierarchical tree is of the form $\mathcal{T} = (V, \Upsilon, v_r, E)$, where:

- V is a finite set of nodes.
- Υ is a mapping that maps each node $v \in V$ to a tree.
- $v_r \in V$ is the root node.
- $E \subseteq V \times V$ is a set of edges.

We now define what it means for an HFTA to *accept* a hierarchical tree:

Definition 4.1. A hierarchical tree $\mathcal{T} = (V, \Upsilon, v_r, E)$ is accepted by an HFTA $\mathcal{H} = (V, \Omega, v_r, \Lambda)$ iff:

- For any node $v \in V$, the tree $\Upsilon(v)$ is accepted by FTA $\Omega(v)$.
- For any edge $(v, v') \in E$, there is an accepting run π of FTA $\Omega(v)$ on tree $\Upsilon(v)$ and an accepting run π' of $\Omega(v')$ on $\Upsilon(v')$ such that there exists a unique leaf node $l \in \Upsilon(v)$ where we have $\pi'\Big(Root(\Upsilon(v'))\Big) \rightarrow \pi(l) \in \Lambda$.

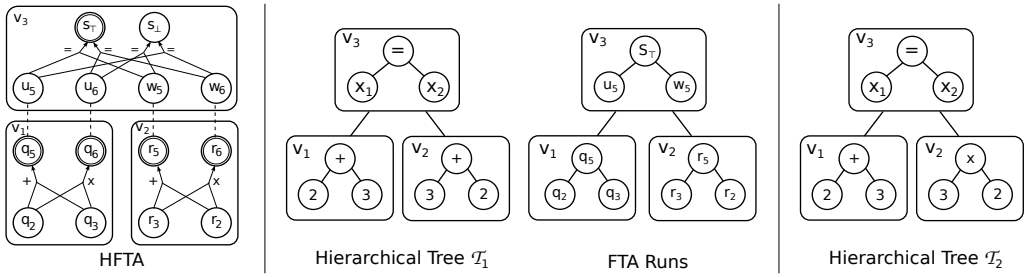

Fig. 5. Example HFTA and hierarchical trees. Left-hand side shows HFTA $\mathcal{H} = (V, \Omega, v_3, \Lambda)$, where nodes $V = \{v_1, v_2, v_3\}$ are represented by rectangles, and annotated FTAs are represented as hypergraphs inside the corresponding rectangles. Specifically, circles in the hypergraph correspond to FTA states, double circles indicate final states, and labeled (hyper-)edges correspond to transitions. Inter-FTA transitions Λ are represented as dashed lines. An example hierarchical tree $\mathcal{T}_1 = (V, \Upsilon, v_3, E)$ that is *accepted* by $\mathcal{H}$ is shown in the middle, where nodes $V = \{v_1, v_2, v_3\}$ are represented by rectangles, the tree annotation $\Upsilon(v_i)$ is depicted inside the corresponding rectangle v_i, and edges are $E = \{(v_3, v_1), (v_3, v_2)\}$. "FTA runs" shows the runs of the FTAs for each v_i in $\mathcal{H}$ on the corresponding tree in $\mathcal{T}_1$. On the right-hand side, we show another example of a hierarchical tree $\mathcal{T}_2$ that is *not accepted* by $\mathcal{H}$.

In other words, a hierarchical tree $\mathcal{T}$ is accepted by an HFTA $\mathcal{H}$ if the tree at each node v of $\mathcal{T}$ is accepted by the corresponding FTA $\Omega(v)$ of $\mathcal{H}$, and the runs of the individual FTAs can be "stitched" together according to the inter-FTA transitions of $\mathcal{H}$. The language of an HFTA $\mathcal{H}$, denoted $\mathcal{L}(\mathcal{H})$, is the set of all hierarchical trees accepted by $\mathcal{H}$.

Example 4.2. Consider the HFTA $\mathcal{H} = (V, \Omega, v_3, \Lambda)$ shown on the left-hand side of Fig. 5. This HFTA has nodes $V = \{v_1, v_2, v_3\}$ annotated with FTAs as follows:

$$\Omega = \{v_1 \mapsto \mathcal{A}_1, v_2 \mapsto \mathcal{A}_3, v_3 \mapsto \mathcal{A}_3\}$$

where:

$\mathcal{A}_1 = (\{q_2, q_3, q_5, q_6\}, \{2, 3, +, \times\}, \{q_5, q_6\}, \Delta_1)$ with transitions Δ_1:

$$2 \to q_2 \quad 3 \to q_3 \quad +(q_2, q_3) \to q_5 \quad \times(q_2, q_3) \to q_6$$

$\mathcal{A}_2 = (\{r_2, r_3, r_5, r_6\}, \{2, 3, +, \times\}, \{r_5, r_6\}, \Delta_2)$ with transitions Δ_2:

$$2 \to r_2 \quad 3 \to r_3 \quad +(r_3, r_2) \to r_5 \quad \times(r_3, r_2) \to r_6$$

$\mathcal{A}_3 = (\{u_5, u_6, w_5, w_6, s_\top, s_\bot\}, \{x_1, x_2, =\}, \{s_\top\}, \Delta_3)$ with transitions Δ_3:

$$\begin{array}{llll} x_1 \to u_5 & x_1 \to u_6 & =(u_5, w_5) \to s_\top & =(u_6, w_5) \to s_\bot \\ x_2 \to w_5 & x_2 \to w_6 & =(u_5, w_6) \to s_\bot & =(u_6, w_6) \to s_\top \end{array}$$

Finally, Λ includes the following four transitions:

$$q_5 \to u_5 \qquad q_6 \to u_6 \qquad r_5 \to w_5 \qquad r_6 \to w_6$$

Intuitively, this HFTA accepts ground first-order formulas of the form $t_1 = t_2$ where t_1 (resp. t_2) is a ground term accepted by $\mathcal{A}_1$ (resp. $\mathcal{A}_2$) such that t_1 and t_2 are equal. For example, $t_1 = 2 + 3$ and $t_2 = 3 + 2$ are collectively accepted. However, $t_1 = 2 + 3$ and $t_2 = 3 \times 2$ are not accepted because t_1 is not equal to t_2. Fig. 5 shows a hierarchical tree $\mathcal{T}_1$ that is accepted by $\mathcal{H}$ and another one $\mathcal{T}_2$ that is rejected by $\mathcal{H}$.

Algorithm 1 Top-level algorithm for inductive relational program synthesis.

1: **procedure** SYNTHESIZE($\mathcal{F}, \mathcal{G}, \Psi$)
input: Function symbols $\mathcal{F} = (f_1, \ldots, f_n)$, grammars $\mathcal{G} = (G_1, \ldots, G_n)$, and specification $\Psi = \forall \vec{x}.\ \phi(\vec{x})$.
output: Mapping $\mathcal{P}$ that maps function symbols in $\mathcal{F}$ to programs.
2: $\mathcal{P} \leftarrow \mathcal{P}_{\text{random}}$;
3: $\Phi \leftarrow \mathit{true}$;
4: **while** *true* **do**
5: **if** $\mathcal{P} = \mathit{null}$ **then return** *null*;
6: **if** Verify($\mathcal{P}, \Psi$) **then return** $\mathcal{P}$;
7: $\Phi \leftarrow \Phi \wedge$ GetRelationalCounterexample($\mathcal{P}, \Psi$);
8: $(\Phi', \mathcal{M}) \leftarrow$ RELAX($\Phi, \mathcal{F}$);
9: $\mathcal{H} \leftarrow$ BUILDHFTA($\Phi', \mathcal{G}, \mathcal{M}$);
10: $\mathcal{P}_\perp \leftarrow \{f_1 \mapsto \perp, \ldots, f_n \mapsto \perp\}$;
11: $\mathcal{P} \leftarrow$ FINDPROGS($\mathcal{H}, \mathcal{P}_\perp, \mathcal{M}$);

5 RELATIONAL PROGRAM SYNTHESIS USING HFTA

In this section, we describe the high-level structure of our relational synthesis algorithm and explain its sub-procedures in the following subsections.

Our top-level synthesis algorithm is shown in Algorithm 1. The procedure SYNTHESIZE takes as input a tuple of function symbols $\mathcal{F} = (f_1, \ldots, f_n)$, their corresponding context-free grammars $\mathcal{G} = (G_1, \ldots, G_n)$, and a relational specification $\Psi = \forall \vec{x}.\ \phi(\vec{x})$. The output is a mapping $\mathcal{P}$ from each function symbol f_i to a program in grammar G_i such that these programs collectively satisfy the given relational specification Ψ, or *null* if no such solution exists.

As mentioned in Section 1, our synthesis algorithm is based on the framework of counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama 2008]. Specifically, given candidate programs $\mathcal{P}$, the verifier checks whether these programs satisfy the relational specification Ψ. If so, the CEGIS loop terminates with $\mathcal{P}$ as a solution (line 6). Otherwise, we obtain a *relational counterexample* in the form of a *ground formula* and use it to strengthen the current *ground relational specification* Φ (line 7). In particular, a relational counterexample can be obtained by instantiating the quantified variables in the relational specification Ψ with concrete inputs that violate Ψ. Then, in the *inductive synthesis phase*, the algorithm finds candidates $\mathcal{P}$ that satisfy the new ground relational specification Φ (lines 8-11) and repeats the aforementioned process. Initially, the candidate programs $\mathcal{P}$ are chosen at random, and the ground relational specification Φ is *true*.

Since the verification procedure is not a contribution of this paper, the rest of this section focuses on the inductive synthesis phase which proceeds in three steps:

(1) ***Relaxation:*** Rather than directly taking Φ as the inductive specification, our algorithm first constructs a relaxation Φ' of Φ (line 8) by replacing each occurrence of function f_i in Φ with a fresh symbol (e.g., $f_1(f_1(1)) = 1$ is converted to $f_{1,1}(f_{1,2}(1)) = 1$). This strategy relaxes the requirement that different occurrences of a function symbol must have the same implementation and allows us to construct our relational version space in a *compositional* way.

(2) ***HFTA construction:*** Given the relaxed specification Φ', the next step is to construct an HFTA whose language consists of exactly those programs that satisfy Φ' (line 9). Our HFTA construction follows the recursive decomposition of Φ' and allows us to compose the individual version spaces of each subterm in a way that the final HFTA is consistent with Φ'.

(3) ***Enforcing functional consistency:*** Since Φ' relaxes the original ground relational specification Φ, the programs for different occurrences of the same function symbol in Φ that are accepted

$$\frac{}{\mathcal{F} \vdash c \hookrightarrow (c, \emptyset)} \text{ (Const)} \qquad \frac{f \in \mathcal{F} \quad f' \text{ is fresh} \quad \mathcal{F} \vdash t_i \hookrightarrow (t_i', \mathcal{M}_i) \quad i = 1, \ldots, m}{\mathcal{F} \vdash f(t_1, \ldots, t_m) \hookrightarrow (f'(t_1', \ldots, t_m'), \Cup_{i=1}^{m} \mathcal{M}_i \Cup \{f \mapsto \{f'\}\})} \text{ (Func)}$$

$$\frac{\mathcal{F} \vdash \psi_1 \hookrightarrow (\psi_1', \mathcal{M}_1) \quad \mathcal{F} \vdash \psi_2 \hookrightarrow (\psi_2', \mathcal{M}_2)}{\mathcal{F} \vdash \psi_1 \; op \; \psi_2 \hookrightarrow (\psi_1' \; op \; \psi_2', \; \mathcal{M}_1 \Cup \mathcal{M}_2)} \text{ (Logical)} \qquad \frac{\mathcal{F} \vdash \Phi \hookrightarrow (\Phi', \mathcal{M})}{\mathcal{F} \vdash \neg\Phi \hookrightarrow (\neg\Phi', \mathcal{M})} \text{ (Neg)}$$

Fig. 6. Inference rules for relaxing inductive relational specification Φ to Φ'. $\mathcal{M}$ maps each function symbol in Φ to its occurrences in Φ'. $\Cup$ is a special union operator for $\mathcal{M}$ and ψ denotes either a term or a formula.

by the HFTA from Step (2) may be different. Thus, in order to guarantee functional consistency, our algorithm looks for programs that are accepted by the HFTA where all occurrences of the same function symbol correspond to the same program (lines 10-11).

The following subsections discuss each of these three steps in more detail.

5.1 Specification Relaxation

Given a ground relational specification Φ and a set of functions $\mathcal{F}$ to be synthesized, the Relax procedure generates a relaxed specification Φ' as well as a mapping $\mathcal{M}$ from each function $f \in \mathcal{F}$ to a *set* of fresh function symbols that represent different occurrences of f in Φ. As mentioned earlier, this relaxation strategy allows us to build a relational version space in a compositional way by composing the individual version spaces for each subterm. In particular, constructing an FTA for a function symbol $f \in \mathcal{F}$ requires knowledge about the possible values of its arguments, which can introduce circular dependencies if we have multiple occurrences of f (e.g., $f(f(1))$). The relaxed specification, however, eliminates such circular dependencies and allows us to build a relational version space in a compositional way.

We present our specification relaxation procedure, Relax, in the form of inference rules shown in Fig. 6. Specifically, Fig. 6 uses judgments of the form $\mathcal{F} \vdash \Phi \hookrightarrow (\Phi', \mathcal{M})$, meaning that ground specification Φ is transformed into Φ' and $\mathcal{M}$ gives the mapping between the function symbols used in Φ and Φ'. In particular, a mapping $f \mapsto \{f_1, \ldots, f_n\}$ in $\mathcal{M}$ indicates that function symbols $f_1, \ldots, f_n$ in Φ' all correspond to the same function f in Φ. It is worthwhile to point out that there are no inference rules for variables in Fig. 6 because specification Φ is a ground formula that does not contain any variables.

Since the relaxation procedure is pretty straightforward, we do not explain the rules in Fig. 6 in detail. At a high level, we recursively transform all subterms and combine the results using a special union operator $\Cup$ defined as follows:

$$(\mathcal{M}_1 \Cup \mathcal{M}_2)(f) = \begin{cases} \emptyset & \text{if } f \notin dom(\mathcal{M}_1) \text{ and } f \notin dom(\mathcal{M}_2) \\ \mathcal{M}_i(f) & \text{if } f \in dom(\mathcal{M}_i) \text{ and } f \notin dom(\mathcal{M}_j) \\ \mathcal{M}_1(f) \cup \mathcal{M}_2(f) & \text{otherwise} \end{cases}$$

In the remainder of this paper, we also use the notation $\mathcal{M}^{-1}$ to denote the inverse of $\mathcal{M}$. That is, $\mathcal{M}^{-1}$ maps each function symbol in the relaxed specification Φ' to a unique function symbol in the original specification Φ (i.e., $f = \mathcal{M}^{-1}(f_i) \Leftrightarrow f_i \in \mathcal{M}(f)$).

5.2 HFTA Construction

We now turn our attention to the relational version space learning algorithm using hierarchical finite tree automata. Given a relaxed relational specification Φ, our algorithm builds an HFTA $\mathcal{H}$ that recognizes exactly those programs that satisfy Φ. Specifically, each node in $\mathcal{H}$ corresponds to a subterm (or subformula) in Φ. For instance, consider the subterm $f(g(1))$ where f and g are

functions to be synthesized. In this case, the HFTA contains a node v_1 for $f(g(1))$ whose child v_2 represents the subterm $g(1)$. The inter-FTA transitions represent data flow from the children subterms to the parent term, and the FTA for each node v is constructed according to the grammar of the top-level constructor of the subterm represented by v.

Fig. 7 describes HFTA construction in more detail using the judgment $\mathcal{G}, \mathcal{M} \vdash \Phi \twoheadrightarrow \mathcal{H}$ where $\mathcal{G}$ describes the DSL for each function symbol [4] and $\mathcal{M}$ is the mapping constructed in Section 5.1. The meaning of this judgment is that, under CFGs $\mathcal{G}$ and mapping $\mathcal{M}$, HFTA $\mathcal{H}$ represents exactly those programs that are consistent with Φ. In addition to the judgment $\mathcal{G}, \mathcal{M} \vdash \Phi \twoheadrightarrow \mathcal{H}$, Fig. 7 also uses an auxiliary judgment of the form $\mathcal{G}, \mathcal{M} \vdash \Phi \rightsquigarrow \mathcal{H}$ to construct the HFTAs for the subformulas and subterms in the specification Φ.

Let us now consider the HFTA construction rules from Fig. 7 in more detail. The first rule, called Const, is the base case of our algorithm and builds an HFTA node for a constant c. Specifically, we introduce a fresh node v_c and construct a trivial FTA $\mathcal{A}$ that accepts only constant c.

The next rule, Func, shows how to build an HFTA for a function term $f(t_1, \ldots, t_m)$. In this rule, we first recursively build the HFTAs $\mathcal{H}_i = (V_i, \Omega_i, v_{r_i}, \Lambda_i)$ for each subterm t_i. Next, we use a procedure called BuildFTA to build an FTA $\mathcal{A}$ for the function symbol f that takes m arguments $x_1, \cdots, x_m$. The BuildFTA procedure is shown in Fig. 8 and is a slight adaptation of the example-guided FTA construction technique described in Section 3.2: Instead of assigning the input example to each argument x_i, our BuildFTA procedure assigns possible values that x_i may take based on the HFTA $\mathcal{H}_i$ for subterm t_i. Specifically, let Q_{f_i} denote the final states of the FTA $\mathcal{A}_{f_i}$ associated with the root node of $\mathcal{H}_i$. If there is a final state q_s^c in Q_{f_i}, this indicates that argument x_i of f may take value c; thus our BuildFTA procedure adds a state $q_{x_i}^c$ in the FTA $\mathcal{A}$ for f (see the Input rule in Fig. 8). The transitions and new states are built exactly as described in Section 3.2 (see the Prod rule of Fig. 8), and we mark every state $q_{s_0}^c$ as a final state if s_0 is the start symbol in f's grammar. To avoid getting an FTA of infinite size, we impose a predefined bound on how many times the Prod rule can be applied when building the FTA. Using this FTA $\mathcal{A}$ for f and the HFTAs $\mathcal{H}_i$ for f's subterms, we now construct a bigger HFTA $\mathcal{H}$ as follows: First, we introduce a new node v_f for function f and annotate it with $\mathcal{A}$. Second, we add appropriate inter-FTA transitions in Λ to account for the binding of formal to actual parameters. Specifically, if q_s^c is a final state of $\mathcal{A}_{f_i}$, we add a transition $q_s^c \rightarrow q_{x_i}^c$ to Λ. Observe that our HFTA is compositional in that there is a separate node for every subterm, and the inter-FTA transitions account for the flow of information from each subterm to its parent.

The next rule, called Logical, shows how to build an HFTA for a subterm $\psi_1\ op\ \psi_2$ where op is either a relation constant (e.g., $=, \leq$) or a binary logical connective (e.g., $\wedge, \leftrightarrow$). As in the previous rule, we first recursively construct the HFTAs $\mathcal{H}_i = (V_i, \Omega_i, v_{r_i}, \Lambda_i)$ for the subterms ψ_1 and ψ_2. Next, we introduce a fresh HFTA node v_{op} for the subterm "$\psi_1\ op\ \psi_2$" and construct its corresponding FTA $\mathcal{A}$ using op's semantics. The inter-FTA transitions between $\mathcal{A}$ and the FTAs annotating the root nodes of $\mathcal{H}_1$ and $\mathcal{H}_2$ are constructed in the same way as in the previous rule. [5] Since the Neg rule for negation is quite similar to the Logical rule, we elide its description in the interest of space.

The last rule, Final, in Fig. 7 simply assigns the final states of the constructed HFTA. Specifically, given specification Φ, we first use the auxiliary judgment $\rightsquigarrow$ to construct the HFTA $\mathcal{H}$ for Φ. However, since we want to accept only those programs that satisfy Φ, we change the final states of the FTA that annotates the root node of $\mathcal{H}$ to be $\{q_{s_0}^\top\}$ rather than $\{q_{s_0}^\top, q_{s_0}^\bot\}$.

[4] If there is an interpreted function in the relational specification, we assume that a trivial grammar with a single production for that function is added to $\mathcal{G}$ by default.

[5] In fact, the Logical and Neg rules could be viewed as special cases of the Func rule where op and $\neg$ have a trivial grammar with a single production. However, we separate these rules for clarity of presentation.

$$\frac{V = \{v_c\} \qquad \mathcal{A} = (\{q_c^{[\![c]\!]}\}, \{c\}, \{q_c^{[\![c]\!]}\}, \{c \to q_c^{[\![c]\!]}\}) \qquad \Omega = \{v_c \mapsto \mathcal{A}\}}{\mathcal{G}, \mathcal{M} \vdash c \rightsquigarrow (V, \Omega, v_c, \emptyset)} \text{ (Const)}$$

$$\frac{\begin{array}{c}\mathcal{G}, \mathcal{M} \vdash t_i \rightsquigarrow (V_i, \Omega_i, v_{r_i}, \Lambda_i) \qquad \Omega_i(v_{r_i}) = (Q_i, \Sigma_i, Q_{f_i}, \Delta_i) \qquad i = 1, \ldots, m \\ \Omega = \bigcup_{i=1}^{m} \Omega_i \cup \{v_f \mapsto \textsc{BuildFTA}(\mathcal{G}(\mathcal{M}^{-1}(f)), [Q_{f_1}, \ldots, Q_{f_m}])\} \\ \Lambda = \bigcup_{i=1}^{m} \Lambda_i \cup \{q_s^c \to q_{x_i}^c \mid q_s^c \in Q_{f_i}, i \in [1, m]\}\end{array}}{\mathcal{G}, \mathcal{M} \vdash f(t_1, \ldots, t_m) \rightsquigarrow (\bigcup_{i=1}^{m} V_i \cup \{v_f\}, \Omega, v_f, \Lambda)} \text{ (Func)}$$

$$\frac{\begin{array}{c}\mathcal{G}, \mathcal{M} \vdash \psi_i \rightsquigarrow (V_i, \Omega_i, v_{r_i}, \Lambda_i) \qquad \Omega_i(v_{r_i}) = (Q_i, \Sigma_i, Q_{f_i}, \Delta_i) \qquad i = 1, 2 \\ Q = \{q_{s_0}^{\perp}, q_{s_0}^{\top}\} \cup \{q_{x_i}^{c_i} \mid q_{s_i}^{c_i} \in Q_{f_i}, i = 1, 2\} \qquad \Delta = \{op(q_{x_1}^{c_1}, q_{x_2}^{c_2}) \to q_{s_0}^{c} \mid c = [\![op]\!](c_1, c_2)\} \\ \Omega = \Omega_1 \cup \Omega_2 \cup \{v_{op} \mapsto (Q, \{x_1, x_2, op\}, \{q_{s_0}^{\perp}, q_{s_0}^{\top}\}, \Delta)\} \qquad \Lambda = \Lambda_1 \cup \Lambda_2 \cup \{q_{s_i}^{c} \to q_{x_i}^{c} \mid q_{s_i}^{c} \in Q_{f_i}, i = 1, 2\}\end{array}}{\mathcal{G}, \mathcal{M} \vdash \psi_1 \ op \ \psi_2 \rightsquigarrow (V_1 \cup V_2 \cup \{v_{op}\}, \Omega, v_{op}, \Lambda)} \text{ (Logical)}$$

$$\frac{\begin{array}{c}\mathcal{G}, \mathcal{M} \vdash \Phi \rightsquigarrow (V, \Omega, v_r, \Lambda) \qquad \Omega(v_r) = (Q, \Sigma, Q_f, \Delta) \\ Q' = \{q_{x_1}^{\perp}, q_{x_1}^{\top}, q_{s_0}^{\perp}, q_{s_0}^{\top}\} \qquad \Delta' = \{\neg(q_{x_1}^{\perp}) \to q_{s_0}^{\top}, \neg(q_{x_1}^{\top}) \to q_{s_0}^{\perp}\} \\ \Omega' = \Omega \cup \{v_{\neg} \mapsto (Q', \{x_1, \neg\}, \{q_{s_0}^{\top}, q_{s_0}^{\perp}\}, \Delta')\} \qquad \Lambda' = \Lambda \cup \{q_s^c \to q_{x_1}^c \mid q_s^c \in Q_f\}\end{array}}{\mathcal{G}, \mathcal{M} \vdash \neg\Phi \rightsquigarrow (V \cup \{v_{\neg}\}, \Omega', v_{\neg}, \Lambda')} \text{ (Neg)}$$

$$\frac{\mathcal{G}, \mathcal{M} \vdash \Phi \rightsquigarrow (V, \Omega, v_r, \Lambda) \qquad \Omega(v_r) = (Q, \Sigma, Q_f, \Delta) \qquad \Omega' = \Omega[v_r \mapsto (Q, \Sigma, \{q_{s_0}^{\top}\}, \Delta)]}{\mathcal{G}, \mathcal{M} \vdash \Phi \twoheadrightarrow (V, \Omega', v_r, \Lambda)} \text{ (Final)}$$

Fig. 7. Rules for constructing HFTA $\mathcal{H} = (V, \Omega, v_r, \Lambda)$ for ground relational specification Φ from grammars $\mathcal{G}$, and function symbol mapping $\mathcal{M}$. Every node v is a fresh node. ψ denotes either a term or a formula. Every variable x_i in function f is assumed to be annotated by f to enforce its global uniqueness.

$$\frac{q_s^c \in Q_i}{G, \vec{Q} \ \vdash \ q_{x_i}^c \in Q, \ x_i \to q_{x_i}^c \in \Delta} \text{ (Input)} \qquad \frac{G, \vec{Q} \vdash q_{s_0}^c \in Q}{G, \vec{Q} \ \vdash \ q_{s_0}^c \in Q_f} \text{ (Output)}$$

$$\frac{(s \to \sigma(s_1, \ldots, s_n)) \in P \quad G, \vec{Q} \vdash q_{s_1}^{c_1} \in Q, \ldots, q_{s_n}^{c_n} \in Q \quad c = [\![\sigma(c_1, \ldots, c_n)]\!]}{G = (T, N, P, s_0), \vec{Q} \ \vdash \ q_s^c \in Q, \ \sigma(q_{s_1}^{c_1}, \ldots, q_{s_n}^{c_n}) \to q_s^c \in \Delta} \text{ (Prod)}$$

Fig. 8. Auxiliary BuildFTA procedure to construct an FTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ for function f that takes m arguments $x_1, \ldots, x_m$. G is a context-free grammar of function f. $\vec{Q} = (Q_1, \ldots, Q_m)$ is a vector where Q_i represents the initial state set for argument x_i.

Example 5.1. Consider the relational specification $\Phi : f(2) = g(f(1))$, where the DSL for f is $e \to x \mid e + 1$ and the DSL for g is $t \to y \mid t \times 2$ (Here, x, y denote the inputs of f and g respectively). We now explain HFTA construction for this specification.

- First, the relaxation procedure transforms the specification Φ to a relaxed version $\Phi' : f_1(2) = g_1(f_2(1))$ and produces a mapping $\mathcal{M} = \{f \mapsto \{f_1, f_2\}, \ g \mapsto \{g_1\}\}$ relating the symbols in Φ, Φ'.
- Fig. 9 shows the HFTA $\mathcal{H}$ constructed for Φ' where we view $+1$ and $\times 2$ as unary functions for ease of illustration. Specifically, by the Const rule of Fig. 7, we build two nodes v_1 and v_2 that correspond to two FTAs that only accept constants 1 and 2, respectively. This sets up the initial state set of FTAs corresponding to f_2 and f_1, which results in two HFTA nodes v_{f_2} and v_{f_1} and their corresponding FTAs $\Omega(v_{f_2})$ and $\Omega(v_{f_1})$ constructed according to the Func rule. (Note that we build the FTAs using only two applications of the Prod rule.) Similarly, we build a node v_{g_1}

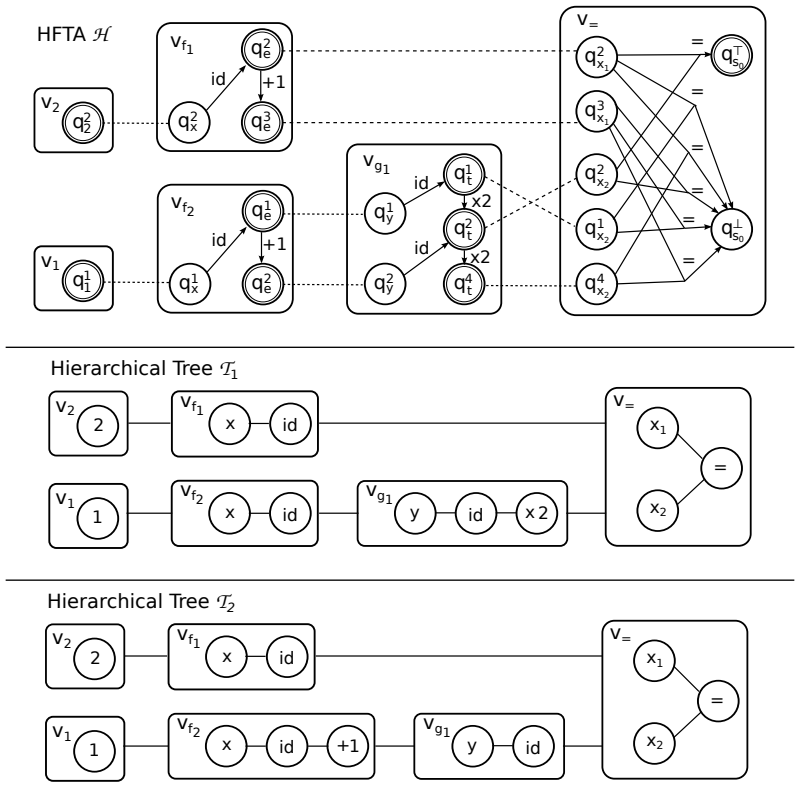

Fig. 9. HFTA $\mathcal{H} = (V, \Omega, v_=, \Lambda)$ for specification $f_1(2) = g_1(f_2(1))$. FTAs are represented as hypergraphs, where circles correspond to FTA states, double circles indicate final states, and labeled (hyper-)edges correspond to transitions. Inter-FTA transitions Λ are represented as dashed lines. Hierarchical trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are both accepted by $\mathcal{H}$.

and FTA $\Omega(v_{g_1})$ by taking the final states of $\Omega(v_{f_2})$ as the initial state set of $\Omega(v_{g_1})$. Finally, the node $v_=$ along with its FTA $\Omega(v_=)$ is built by the Logical rule.

- Fig. 9 also shows two hierarchical trees, $\mathcal{T}_1$ and $\mathcal{T}_2$, that are accepted by the HFTA $\mathcal{H}$ constructed above. However, note that only $\mathcal{T}_1$ corresponds to a valid solution to the original synthesis problem because (1) in $\mathcal{T}_1$ both f_1 and f_2 refer to the program $f = \lambda x.\ x$, and (2) in $\mathcal{T}_2$ these two occurrences (i.e., f_1 and f_2) of f correspond to different programs.

Theorem 5.2. **(HFTA Soundness)** *Suppose $\mathcal{G}, \mathcal{M} \vdash \Phi \twoheadrightarrow \mathcal{H}$ according to the rules from Fig 7, and let $f_1, \ldots, f_n$ be the function symbols used in the relaxed specification Φ. Given a hierarchical tree $\mathcal{T} = (V, \Upsilon, v_r, E)$ that is accepted by $\mathcal{H}$, we have:*

(1) $\Upsilon(v_{f_i})$ is a program that conforms to grammar $\mathcal{G}(\mathcal{M}^{-1}(f_i))$
(2) $\Upsilon(v_{f_1}), \ldots, \Upsilon(v_{f_n})$ satisfy Φ, i.e. $\Upsilon(v_{f_1}), \ldots, \Upsilon(v_{f_n}) \models \Phi$

Proof. See Appendix A of the extended version [Wang et al. 2018c]. □

Theorem 5.3. **(HFTA Completeness)** *Let Φ be a ground formula where every function symbol $f_1, \ldots, f_n$ occurs exactly once, and suppose we have $\mathcal{G}, \mathcal{M} \vdash \Phi \twoheadrightarrow \mathcal{H}$. If there are implementations P_i of f_i such that $P_1, \ldots, P_n \models \Phi$, where P_i conforms to grammar $\mathcal{G}(\mathcal{M}^{-1}(f_i))$, then there exists a hierarchical tree $\mathcal{T} = (V, \Upsilon, v_r, E)$ accepted by $\mathcal{H}$ such that $\Upsilon(v_{f_i}) = P_i$ for all $i \in [1, n]$.*

Proof. See Appendix A of the extended version [Wang et al. 2018c]. □

Algorithm 2 Algorithm for enforcing functional consistency.

1: **procedure** FindProgs($\mathcal{H}, \mathcal{P}, \mathcal{M}$)

Input: HFTA $\mathcal{H} = (V, \Omega, v_r, \Lambda)$, mapping $\mathcal{P}$ that maps a function symbol to a program, and mapping $\mathcal{M}$ that maps a function symbol to its occurrences.
Output: Mapping $\mathcal{P}_{\text{res}}$ that maps each function symbol to a program.

2: $f \leftarrow$ ChooseUnassigned($\mathcal{P}$);
3: **if** $f = null$ **then return** $\mathcal{P}$;
4: $f_i \leftarrow$ ChooseOccurrence($\mathcal{M}, f$);
5: **for each** $p \in \left\{ \Upsilon(v_{f_i}) \;\middle|\; (V, \Upsilon, v_r, E) \in \mathcal{L}(\mathcal{H}),\ v_{f_i} \in V \right\}$ **do**
6: $\mathcal{H}' \leftarrow$ Propagate($\mathcal{H}, p, f, \mathcal{M}$);
7: **if** $\mathcal{L}(\mathcal{H}') = \emptyset$ **then continue**;
8: $\mathcal{P}' \leftarrow \mathcal{P}[f \mapsto p]$;
9: $\mathcal{P}_{\text{res}} \leftarrow$ FindProgs($\mathcal{H}', \mathcal{P}', \mathcal{M}$);
10: **if** $\mathcal{P}_{\text{res}} \neq null$ **then return** $\mathcal{P}_{\text{res}}$;
11: **return** *null*;

5.3 Enforcing Functional Consistency

As stated by Theorems 5.2 and 5.3, the HFTA method discussed in the previous subsection gives a sound and complete synthesis procedure with respect to the *relaxed* specification where each function symbol occurs exactly once. However, a hierarchical tree $\mathcal{T}$ that is accepted by the HFTA might still violate functional consistency by assigning different programs to the same function f used in the original specification. In this section, we describe an algorithm for finding hierarchical trees that (a) are accepted by the HFTA and (b) conform to the functional consistency requirement.

Algorithm 2 describes our technique for finding accepting programs that obey functional consistency. Given an HFTA $\mathcal{H}$ and a mapping $\mathcal{M}$ from function symbols in the original specification to those in the relaxed specification, the FindProgs procedure finds a mapping $\mathcal{P}_{\text{res}}$ from each function symbol f_i in the domain of $\mathcal{M}$ to a program p_i such that $p_1, \ldots, p_n$ are accepted by $\mathcal{H}$. Since the resulting mapping $\mathcal{P}_{\text{res}}$ maps each function symbol in the *original* specification to a single program, the solution returned by FindProgs is guaranteed to be a valid solution for the original relational synthesis problem.

We now explain how Algorithm 2 works in more detail. At a high level, FindProgs is a recursive procedure that, in each invocation, updates the current mapping $\mathcal{P}$ by finding a program for a currently unassigned function symbol f. In particular, we say that a function symbol f is unassigned if $\mathcal{P}$ maps f to $\perp$. Initially, all function symbols are unassigned, but FindProgs iteratively makes assignments to each function symbol in the domain of $\mathcal{P}$. Eventually, if all function symbols have been assigned, this means that $\mathcal{P}$ is a valid solution; thus, Algorithm 2 returns $\mathcal{P}$ at line 3 if ChooseUnassigned yields *null*.

Given a function symbol f that is currently unassigned, FindProgs first chooses some occurrence f_i of f in the relaxed specification (i.e., $f_i \in \mathcal{M}(f)$) and finds a program p for f_i. In particular, given a hierarchical tree $\mathcal{T}$ accepted by $\mathcal{H}$ [6], we find the program p that corresponds to f_i in $\mathcal{T}$ (line 5). Now, since every occurrence of f must correspond to the same program, we use a procedure called Propagate that propagates p to all other occurrences of f (line 6). The procedure Propagate is shown in Algorithm 3 and returns a modified HFTA $\mathcal{H}'$ that enforces that all occurrences of f are consistent with p. (We will discuss the Propagate procedure in more detail after finishing the discussion of FindProgs).

[6] Section 6 describes a strategy for lazily enumerating hierarchical trees accepted by a given HFTA.

Algorithm 3 Auxilary procedure for propagating a program in HFTA.

1: **procedure** PROPAGATE($\mathcal{H}, p, f, \mathcal{M}$)

Input: HFTA $\mathcal{H} = (V, \Omega, v_r, \Lambda)$, program p, function symbol f, $\mathcal{M}$ maps function symbols to occurrences
Output: An updated HFTA $\mathcal{H}' = (V, \Omega', v_r, \Lambda')$

2: $\Omega' \leftarrow \Omega, \quad \Lambda' \leftarrow \Lambda$;
3: **for each** $v \in \{v_{f_i} \mid f_i \in \mathcal{M}(f), v_{f_i} \in V\}$ **do**
4: $(Q, \Sigma, Q_f, \Delta) \leftarrow \Omega(v)$;
5: $Q'_f \leftarrow \text{ReachableFinalStates}(\Omega(v), p)$; ▷ remove unreachable final states
6: $\Omega' \leftarrow \Omega'[v \mapsto (Q, \Sigma, Q'_f, \Delta)]$;
7: $\Lambda' \leftarrow \Lambda' \setminus \cup_{q \in Q_f \setminus Q'_f} \{q \rightarrow q' \mid q \rightarrow q' \in \Lambda'\}$; ▷ remove spurious transitions
8: **return** $(V, \Omega', v_r, \Lambda')$;

The loop in lines 5–10 of Algorithm 2 performs backtracking search. In particular, if the language of $\mathcal{H}'$ becomes empty after the call to PROPAGATE, this means that p is not a suitable implementation of f — i.e., given the current mapping $\mathcal{P}$, there is no extension of $\mathcal{P}$ where p is assigned to f. Thus, the algorithm backtracks at line 7 by moving on to the next program for f.

On the other hand, assuming that the call to PROPAGATE does not result in a contradiction (i.e., $\mathcal{L}(\mathcal{H}') \neq \emptyset$), we try to find an implementation of the remaining function symbols via the recursive call to FINDPROGS at line 9. If the recursive call does not yield *null*, we have found a set of programs that both satisfy the relational specification and obey functional consistency; thus, we return $\mathcal{P}_{\text{res}}$ at line 10. Otherwise, we again backtrack and look for a different implementation of f.

Propagate subroutine. We now discuss the PROPAGATE subroutine for enforcing that different occurrences of a function have the same implementation. Given an HFTA $\mathcal{H}$ and a function symbol f with candidate implementation p, PROPAGATE returns a new HFTA $\mathcal{H}'$ such that the FTAs for all occurrences of f only accept programs that have the same input-output behavior as p.

In more detail, the loop in lines 3–7 of Algorithm 3 iterates over all HFTA nodes v that correspond to some occurrence of f. Since we want to make sure that the FTA for v only accepts those programs that have the same input-output behavior as p, we first compute all final states Q'_f that can be reached by successful runs of the FTA on program p (line 5) and change the final states of $\Omega(v)$ to Q'_f (line 6). Since some of the inter-FTA transitions become spurious as a result of this modification, we also remove inter-FTA transitions $q \rightarrow q'$ where q is no longer a final state of its corresponding FTA (line 7). These modifications ensure that the FTAs for different occurrences of f *only* accept programs that have the same behavior as p.

Example 5.4. We now illustrate how FINDPROGS extracts programs from the HFTA in Example 5.1. Suppose we first choose occurrence f_2 of function f and its implementation $f_2 = \lambda x.\ x + 1$. Since f_1 and f_2 must have the same implementation, the call to PROPAGATE results in the modified HFTA $\mathcal{H}_2$ shown in Fig. 10. However, observe that $\mathcal{L}(\mathcal{H}_2)$ is empty because the final state $q^{\top}_{s_0}$ in FTA $\Omega(v_{=})$ is only reachable via state $q^2_{x_1}$, but $q^2_{x_1}$ no longer has incoming transitions. Thus, the algorithm backtracks to the only other implementation choice for f_2, namely $\lambda x.x$, and PROPAGATE now yields the HFTA $\mathcal{H}_3$ from Fig. 10. This time, $\mathcal{L}(\mathcal{H}_3)$ is not empty; hence, the algorithm moves on to function symbol g. Since the only hierarchical tree in $\mathcal{L}(\mathcal{H}_3)$ is $\mathcal{T}_1$ from Fig. 9, we are forced to choose the implementation $g = \lambda y.\ y \times 2$. Thus, the FINDPROGS procedure successfully returns $\mathcal{P} = \{f \mapsto \lambda x.x,\ g \mapsto \lambda y.\ y \times 2\}$.

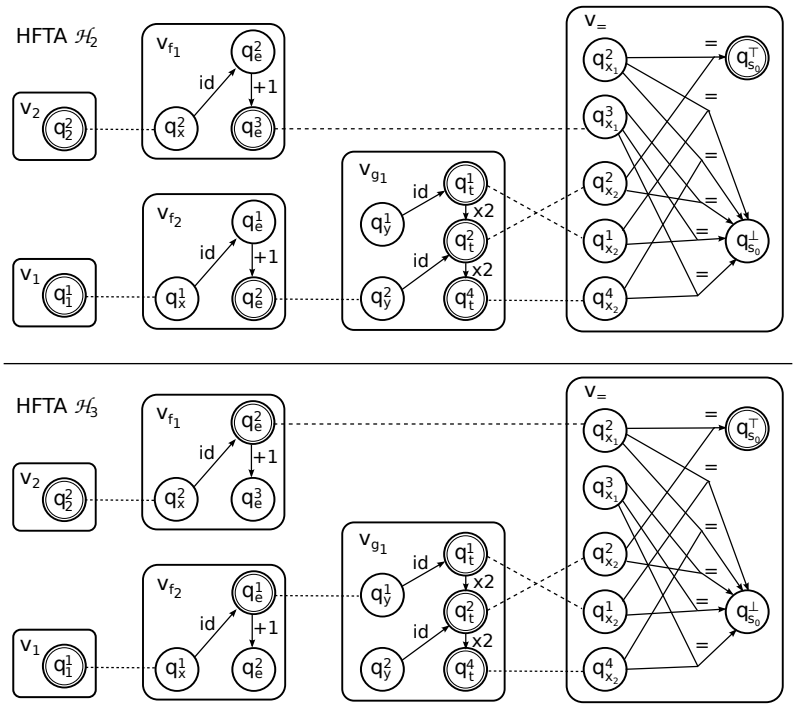

Fig. 10. HFTAs after propagation. $\mathcal{H}_2$ is obtained by propagating $f_1 = f_2 = \lambda x.\, x{+}1$ on HFTA $\mathcal{H}$ in Example 5.1 (shown in Fig. 9). $\mathcal{H}_3$ is obtained by propagating $f_1 = f_2 = \lambda x.\, x$ on HFTA $\mathcal{H}$.

5.4 Properties of the Synthesis Algorithm

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

Theorem 5.5. **(Soundness)** *Assuming the soundness of the Verify procedure invoked by Algorithm 1, then the set of programs $\mathcal{P}$ returned by* Synthesize *are guaranteed to satisfy the relational specification Ψ, and program P_i of function f_i conforms to grammar $\mathcal{G}(f_i)$.*

Proof. See Appendix A of the extended version [Wang et al. 2018c]. □

Theorem 5.6. **(Completeness)** *If there is a set of programs that (a) satisfy Ψ and (b) can be implemented using the DSLs given by $\mathcal{G}$, then Algorithm 1 will eventually terminate with programs $\mathcal{P}$ satisfying Ψ.*

Proof. See Appendix A of the extended version [Wang et al. 2018c]. □

6 IMPLEMENTATION

We have implemented the proposed relational program synthesis approach in a framework called Relish. To demonstrate the capabilities of the Relish framework, we instantiate it on the first two domains from Section 2.2, namely string encoders/decoders, and comparators. In this section, we describe our implementation of Relish and its instantiations.

6.1 Implementation of Relish Framework

While the implementation of the Relish framework closely follows the algorithm described in Section 5, it performs several important optimizations that we discuss next.

$$
\begin{array}{llll}
\textit{EncodedText} & E & := & M \mid \textit{padToMultiple}(E, \textit{num}, \textit{char}) \mid \textit{header}(E) \\
\textit{Mapper} & M & := & \textit{enc16}(B) \mid \textit{enc32}(B) \mid \textit{enc32Hex}(B) \mid \textit{enc64}(B) \mid \textit{enc64XML}(B) \mid \textit{encUU}(B) \\
\textit{ByteArray} & B & := & x \mid \textit{reshape}(B, \textit{num}) \mid \textit{encUTF8}(I) \mid \textit{encUTF16}(I) \mid \textit{encUTF32}(I) \\
\textit{IntArray} & I & := & \textit{codePoint}(x)
\end{array}
$$

$$x \in \textit{Variable} \qquad \textit{num} \in \{1, 2, \ldots, 8\} \qquad \textit{char} \in \{\text{`='}, \text{`'}, \ldots\}$$

(a) Context-free grammar for encoders.

$$
\begin{array}{llll}
\textit{DecodedData} & D & := & B \mid \textit{asUnicode}(I) \\
\textit{IntArray} & I & := & \textit{decUTF8}(B) \mid \textit{decUTF16}(B) \mid \textit{decUTF32}(B) \\
\textit{ByteArray} & B & := & M \mid \textit{invReshape}(B, \textit{num}) \\
\textit{Mapper} & M & := & \textit{dec16}(C) \mid \textit{dec32}(C) \mid \textit{dec32Hex}(C) \mid \textit{dec64}(C) \mid \textit{dec64XML}(C) \mid \textit{decUU}(C) \\
\textit{CharArray} & C & := & x \mid \textit{removePad}(C, \textit{char}) \mid \textit{substr}(C, \textit{num})
\end{array}
$$

$$x \in \textit{Variable} \qquad \textit{num} \in \{1, 2, \ldots, 8\} \qquad \textit{char} \in \{\text{`='}, \text{`'}, \ldots\}$$

(b) Context-free grammar for decoders.

Fig. 11. Context-free grammars for Unicode string encoders and decoders.

Lazy enumeration. Recall that Algorithm 2 needs to (lazily) enumerate all hierarchical trees accepted by a given HFTA. Furthermore, since we want to find a program with the greatest generalization power, our algorithm should enumerate more promising programs first. Based on these criteria, we need a mechanism for predicting the generalization power of a hierarchical tree using some heuristic cost metric. In our implementation, we associate a non-negative cost with every DSL construct and compute the cost of a given hierarchical tree by summing up the costs of all its nodes. Because hierarchical trees with lower cost are likely to have better generalization power, our algorithm lazily enumerates hierarchical trees according to their cost.

In our implementation, we reduce the problem of enumerating hierarchical trees accepted by an HFTA to the task of enumerating B-paths in a hypergraph [Gallo et al. 1993]. In particular, we first flatten the HFTA into a standard FTA by combining the individual tree automata at each node via the inter-FTA transitions. We then represent the resulting flattened FTA as a hypergraph where the FTA states correspond to nodes and a transition $\sigma(q_1, \ldots, q_n) \rightarrow q$ corresponds to a B-edge $(\{q_1, \ldots, q_n\}, q)$ with weight $\textit{cost}(\sigma)$. Given this representation, the problem of finding the lowest-cost hierarchical tree accepted by an HFTA becomes equivalent to the task of finding a minimum weighted B-path in a hypergraph, and our implementation leverages known algorithms for solving this problem [Gallo et al. 1993].

Verification. Because our overall approach is based on the CEGIS paradigm, we need a separate verification step to both check the correctness of the programs returned by the inductive synthesizer and find counterexamples if necessary. However, because heavy-weight verification can add considerable overhead to the CEGIS loop, we *test* the program against a large set of inputs rather than performing full-fledged verification in each iteration. Specifically, we generate a set of validation inputs by computing all possible permutations of a finite set up to a bounded length k and check the correctness of the candidate program against this validation set. In our experiments, we use over a million test cases in each iteration, and resort to full verification only when the synthesized program passes all of these test cases.

6.2 Instantiation for String Encoders and Decoders

While Relish is a generic framework that can be used in various application domains, one needs to construct suitable DSLs and write relational specifications for each different domain. In this section, we discuss our instantiation of Relish for synthesizing string encoders and decoders. Since we

have already explained the relational property of interest in Section 2.2, we discuss two simple DSLs, presented in Fig. 11, for implementing encoders and decoders respectively.

DSL for encoders. We designed a DSL for string encoders by reviewing several different encoding mechanisms and identifying their common building blocks. Specifically, this DSL allows transforming a Unicode string (or binary data) to a sequence of restricted ASCII characters (e.g., to fulfill various requirements of text-based network transmission protocols). At a high-level, programs in this DSL first transform the input string to an integer array and then to a byte array. The encoded text is obtained by applying various kinds of mappers to the byte array, padding it, and attaching length information. In what follows, we informally describe the semantics of the constructs used in the encoder DSL.

- ***Code point representation:*** The *codePoint* function converts string s to an integer array I where $I[j]$ corresponds to the Unicode code point for $s[j]$.
- ***Mappers:*** The *encUTF8/16/32* functions transform the code point array to a byte array according to the corresponding standards of UTF-8, UTF-16, UTF-32, respectively. The other mappers *encX* can further transform the byte array into a sequence of restricted ASCII characters based on different criteria. For example, the *enc16* function is a simple hexadecimal mapper that can convert binary data `0x6E` to the string "6E".
- ***Padding:*** The *padToMultiple* function takes an existing character array E and pads it with a sequence of extra *char* characters to ensure that the length of the padded sequence is evenly divisible by *num*.
- ***Header:*** The *header(E)* function prepends the ASCII representation of the length of the text to E.
- ***Reshaping:*** The function *reshape(B, num)* first concatenates all bytes in B, then regroups them such that each group only contains *num* bits (instead of 8), and finally generates a new byte array where each byte is equal to the value of corresponding group. For example, *reshape*`([0xFF],4) = [0x0F,0x0F]` and *reshape*`([0xFE],2) = [0x03,0x03,0x03,0x02]`.

DSL for decoders. The decoder DSL is quite similar to the encoder one and is structured as follows: Given an input string x, programs in this DSL first process x by removing the header and/or padding characters and then transform it to a byte array using a set of pre-defined mappers. The decoded data can be either the resulting byte array or a Unicode string obtained by converting the byte array to an integer Unicode code point array. In more detail, the decoder DSL supports the following built-in operators:

- ***Unicode conversion:*** The *asUnicode* function converts an integer array I to a string s, where $s[j]$ corresponds to the Unicode symbol of code point $I[j]$.
- ***Mappers:*** The *decUTF8/16/32* functions transform a byte array to a code point array based on standards of UTF-8, UTF-16, UTF-32, respectively. The other mappers *decX* transform a sequence of ASCII characters to byte arrays according to their standard transformation rules.
- ***Character removal:*** The *removePad* function removes all trailing characters *char* from a given string C. The *substr* function takes a string C and an index *num*, and returns the sub-string of C from index *num* to the end.
- ***Reshaping:*** The *invReshape* function takes a byte array B, collects *num* bits from the least significant end of each byte, concatenates the bit sequence and regroups every eight bits to generate a new byte array. For instance, *invReshape*`([0x0E,0x0F],4) = [0xEF]`.

Example 6.1. The desired `encode/decode` functions from Example 2.3 can be implemented in our DSL as follows:

```
encode(x) : padToMultiple (enc64 (reshape (encUTF8 (codePoint (x)), 6)), 4, '=')
decode(x) : asUnicode (decUTF8 (invReshape (dec64 (removePad (x, '=')), 6)))
```

$$
\begin{array}{llll}
\textit{Comparator} & C & := & B \mid \textit{chain}(B, C) \\
\textit{Basic} & B & := & \textit{intCompare}(I_1, I_2) \mid \textit{strCompare}(S_1, S_2) \\
\textit{Integer} & I & := & \textit{countChar}(S, c) \mid \textit{length}(S) \mid \textit{toInt}(S) \\
\textit{String} & S & := & \textit{substr}(v, P_1, P_2) \\
\textit{Position} & P & := & \textit{pos}(v, t, k, d) \mid \textit{constPos}(k)
\end{array}
$$

$$
v \in \{\, x, y \,\} \qquad c \in \textit{Characters} \qquad t \in \textit{Tokens} \qquad k \in \textit{Integers} \qquad d \in \{\, \textit{Start}, \textit{End} \,\}
$$

Fig. 12. Context-free grammar for comparators.

6.3 Instantiation for Comparators

We have also instantiated the Relish framework to enable automatic generation of custom string comparators. As described in Section 2.2, this domain is an interesting ground for relational program synthesis because comparators must satisfy three different relational properties (i.e., anti-symmetry, transitivity, and totality). In what follows, we describe the DSL from Fig. 12 that Relish uses to synthesize these comparators.

In more detail, programs in our comparator DSL take as input a pair of strings x, y, and return -1, 0, or 1 indicating that x precedes, is equal to, or succeeds y respectively. Specifically, a program is either a *basic comparator* B or a *comparator chain* of the form $\textit{chain}(B_1, \ldots, B_n)$ which returns the result of the *first* comparator that does not evaluate to zero. The DSL allows two basic comparators, namely *intCompare* and *strCompare*. The integer or string inputs to these basic comparators can be obtained using the following functions:

- ***Substring extraction:*** The *substring* function is used to extract substrings of the input string. In particular, for a string v and positions P_1, P_2, it returns the substring of v that starts at index P_1 and ends at index P_2.
- ***Position identifiers:*** A position P can either be a constant index ($\textit{constPos}(k)$) or the (start or end) index of the k'th occurrence of the match of token t in the input string ($\textit{pos}(v, t, k, d)$). [7] For example, we have $\textit{pos}(\text{“}12ab\text{”}, \texttt{Number}, 1, \texttt{Start}) = 0$ and $\textit{pos}(\text{“}12ab\text{”}, \texttt{Number}, 1, \texttt{End}) = 2$ where `Number` is a token indicating a sequence of digits.
- ***Numeric string features:*** The DSL allows extracting various numeric features of a given string S. In particular, *countChar* yields the number of occurrences of a given character c in string S, *length* yields string length, and *toInt* converts a string representing an integer to an actual integer (i.e., $\textit{toInt}(\text{“}123\text{”}) = 123$ but $\textit{toInt}(\text{“abc"})$ throws an exception).

Example 6.2. Consider Example 2.4 where the user wants to sort integers based on the number of occurrences of the number 5, and, in the case of a tie, sort them based on the actual integer values. This functionality can be implemented by the following simple program in our DSL:

```
chain (intCompare (countChar (x, '5'), countChar (y, '5')),
       intCompare (toInt (x), toInt (y)) )
```

7 EVALUATION

We evaluate Relish by using it to automatically synthesize (1) string encoders and decoders for program inversion tasks collected from prior work [Hu and D'Antoni 2017] and (2) string comparators to solve sorting problems posted on StackOverflow. The goal of our evaluation is to answer the following questions:

- How does Relish perform on various relational synthesis tasks from two domains?
- What is the benefit of using HFTAs for relational program synthesis?

[7] Tokens are chosen from a predefined universe of regular expressions.

Experimental setup. To evaluate the benefit of our approach over a base-line, we compare our method against EUSolver [Alur et al. 2017], an enumeration-based synthesizer that won the General Track of the most recent SyGuS competition [Alur et al. 2013]. Since EUSolver only supports synthesis tasks in linear integer arithmetic, bitvectors, and basic string manipulations by default, we extend it to encoder/decoders and comparators by implementing the same DSLs described in Section 6. Additionally, we implement the same CEGIS loop used in Relish for EUSolver and use the same bounded verifier in our evaluation. All experiments are conducted on a machine with Intel Xeon(R) E5-1620 v3 CPU and 32GB of physical memory, running the Ubuntu 14.04 operating system. Due to finite computational resources, we set a time limit of 24 hours for each benchmark.

7.1 Results for String Encoders and Decoders

In our first experiment, we evaluate Relish by using it to simultaneously synthesize Unicode string encoders and decoders, which are required to be inverses of each other.

Benchmarks. We collect a total of ten encoder/decoder benchmarks, seven of which are taken from a prior paper on program inversion [Hu and D'Antoni 2017]. Since the 14 benchmarks from prior paper [Hu and D'Antoni 2017] are essentially seven pairs of encoders and decoders, we have covered all their encoder/decoder benchmarks. The remaining three benchmarks, namely, *Base32hex, UTF-32, and UTF-7*, are also well-known encodings. Unlike previous work on program inversion, our goal is to solve the considerably more difficult problem of simultaneously synthesizing the encoder *and* decoder from input-output examples rather than inverting an existing function. For each benchmark, we use 2-3 input-output examples taken from the documentation of the corresponding encoders. We also specify the relational property $\forall x$. `decode(encode(x))=x` and use the encoder/decoder DSLs presented in Section 6.2.

Main results. Our main experimental results are summarized in Table 1, where the first two columns (namely, "Enc Size" and "Dec Size") describe the size of the target program in terms of the number of AST nodes. [8] The next three columns under **Relish** summarize the results obtained by running Relish on each of these benchmarks, and the three columns under **EUSolver** report the same results for EUSolver. Specifically, the column labeled "Iters" shows the total number of iterations inside the CEGIS loop, "Total" shows the total synthesis time in seconds, and "Synth" indicates the time (in seconds) taken by the inductive synthesizer (i.e., excluding verification). If a tool fails to solve the desired task within the 24 hour time limit, we write T/O to indicate a time-out. Finally, the last column labeled "Speed-up" shows the speed-up of Relish over EUSolver for those benchmarks where neither tool times out.

As shown in Table 1, Relish can correctly solve all of these benchmarks [9] and takes an average of 17.9 seconds per benchmark. In contrast, EUSolver solves half of the benchmarks within the 24 hour time limit and takes an average of approximately 12 *minutes* per benchmark that it is able to solve. For the five benchmarks that can be solved by both tools, the average speed-up of Relish over EUSolver is 46.5x. [10] These statistics clearly demonstrate the advantages of our HFTA-based approach compared to enumerative search: Even though both tools use the same DSLs, verifier, and CEGIS architecture, Relish unequivocally outperforms EUSolver across all benchmarks.

Next, we compare Relish and EUSolver in terms of the number of CEGIS iterations for the five benchmarks that can be solved by both tools. As we can see from Table 1, Relish takes 2.8 CEGIS iterations on average, whereas EUSolver needs an average of 3.4 attempts to find the correct

[8] We obtain this information by manually writing a simplest DSL program for achieving the desired task.
[9] We manually inspected the synthesized solutions and confirmed their correctness.
[10] Here, we use geometric mean to compute the average since the arithmetic mean is not meaningful for ratios.

Table 1. Experimental results on Unicode string encoders and decoders.

Benchmark	Enc Size	Dec Size	Relish				EUSolver			Speedup
			Iters	Total(s)	Synth(s)	Mem(MB)	Iters	Total(s)	Synth(s)	
Base16	6	6	3	16.2	10.4	551	3	494.2	489.3	30.4x
Base32	9	8	5	21.0	14.6	458	-	T/O	T/O	-
Base32hex	9	8	5	22.9	15.6	468	-	T/O	T/O	-
Base64	9	8	5	16.4	8.8	916	-	T/O	T/O	-
Base64xml	9	8	6	23.4	15.5	1843	-	T/O	T/O	-
UU	10	10	5	23.3	15.2	843	-	T/O	T/O	-
UTF-8	6	6	4	17.7	11.9	916	4	536.4	531.8	30.2x
UTF-16	6	6	2	11.5	5.4	285	4	1265.2	1259.9	110.4x
UTF-32	6	6	2	11.6	4.8	284	3	771.1	765.1	66.4x
UTF-7	6	6	3	14.8	9.2	285	3	475.2	470.1	32.1x
Average	7.6	7.2	4.0	17.9	11.1	685	3.4	708.4	703.2	46.5x

program. This discrepancy suggests that our HFTA-based method might have better generalization power compared to enumerative search. In particular, our method first generates a version space that contains *all* tuples of programs that satisfy the relational specification and then searches for the best program in this version space. In contrast, EUSolver returns the *first* program that satisfies the current set of counterexamples; however, this program may not be the best (i.e., lowest-cost) one in Relish's version space.

Finally, we compare Relish against EUSolver in terms of synthesis time per CEGIS iteration: Relish takes an average of 4.5 seconds per iteration, whereas EUSolver takes 208.4 seconds on average across the five benchmarks that it is able to solve. To summarize, these results clearly indicate the advantages of our approach compared to enumerative search when synthesizing string encoders and decoders.

Memory usage. We now investigate the memory usage of Relish on the encoder/decoder benchmarks (see column labeled Mem in Table 1). Here, the memory usage varies between 284MB and 1843MB, with the average being 685MB. As we can see from Table 1, the memory usage mainly depends on (1) the number of CEGIS iterations and (2) the size of programs to be synthesized. Specifically, as the CEGIS loop progresses, the number of function occurrences in the ground relational specification increases, which results in larger HFTAs. For example, the impact of the number of CEGIS iterations on memory usage becomes apparent by comparing the "Base64xml" and "UTF-32" benchmarks, which take 6 and 2 iterations and consume 1843 MB and 284 MB respectively. In addition to the number of CEGIS iterations, the size of the target program also has an impact on memory usage. Intuitively, the larger the synthesized programs, the larger the size of the individual FTAs; thus, memory usage tends to increase with program size. For example, the impact of program size on memory usage is illustrated by the difference between the "UU" and "Base32" benchmarks.

7.2 Results for String Comparators

In our second experiment, we evaluate Relish by using it to synthesize string comparators for sorting problems obtained from StackOverflow. Even though our goal is to synthesize a single `compare` function, this problem is still a relational synthesis task because the generated program must obey two 2-safety properties (i.e., anti-symmetry and totality) and one 3-safety property (i.e., transitivity). Thus, we believe that comparator synthesis is also an interesting and relevant test-bed for evaluating relational synthesizers.

Benchmarks. To perform our evaluation, we collected 20 benchmarks from StackOverflow using the following methodology: First, we searched StackOverflow for the keywords "*Java string*

Table 2. Experimental results on comparators.

Benchmark	Size	Relish				EUSolver			Speedup
		Iters	Total(s)	Synth(s)	Mem(MB)	Iters	Total(s)	Synth(s)	
comparator-1	5	2	2.0	0.1	12	2	2.6	0.7	1.3x
comparator-2	13	5	3.4	0.2	1375	5	5.5	2.5	1.6x
comparator-3	13	5	4.1	0.5	385	6	21.5	18.5	5.3x
comparator-4	23	6	7.7	1.4	1401	10	299.7	291.9	38.8x
comparator-5	21	7	8.8	1.2	548	10	650.9	643.3	74.1x
comparator-6	23	4	9.1	0.2	30	11	57.3	52.5	6.3x
comparator-7	25	7	9.8	0.6	1386	9	640.8	632.5	65.1x
comparator-8	23	6	9.9	1.6	117	7	50.7	45.2	5.1x
comparator-9	41	10	25.5	12.7	2146	-	T/O	T/O	-
comparator-10	21	11	40.7	32.4	1454	12	1301.9	1295.3	32.0x
comparator-11	41	9	42.9	7.4	4507	13	13966.0	13952.7	325.2x
comparator-12	17	10	75.2	66.7	1045	15	9171.0	9161.4	122.0x
comparator-13	27	10	85.0	60.1	3834	13	10742.8	10736.3	126.3x
comparator-14	25	8	102.5	93.2	2022	-	T/O	T/O	-
comparator-15	17	9	116.7	112.3	1622	16	26443.4	26434.6	226.5x
comparator-16	19	7	130.1	124.2	1383	-	T/O	T/O	-
comparator-17	43	11	196.2	183.4	4460	-	T/O	T/O	-
comparator-18	41	10	406.5	351.0	18184	-	T/O	T/O	-
comparator-19	65	9	523.9	485.7	18367	-	T/O	T/O	-
comparator-20	-	-	T/O	T/O	-	-	T/O	T/O	-
Average	26.5	7.7	94.7	80.8	3382	9.9	4873.4	4866.7	26.0x

comparator". Then, we manually inspected each of these posts and retained exactly those that satisfy the following criteria:

- The question in the post should involve writing a `compare` function for sorting strings.
- The post should contain a list of sample strings that are sorted in the desired way.
- The post should contain a natural language description of the desired sorting task.

The relational specification Ψ for each benchmark consists of the following three parts:

- Universally-quantified formulas reflecting the three relational properties that `compare` has to satisfy (i.e., anti-symmetry, transitivity, and totality).
- Another quantified formula that stipulates reflexivity (i.e., $\forall x.\ \texttt{compare}(x, x) = 0$)
- Quantifier-free formulas that correspond to the input-output examples from the StackOverflow post. In particular, given a sorted list l, if string x appears before string y in l, we add the examples `compare(x,y) = -1` and `compare(y,x) = 1`.

Among these benchmarks, the number of examples range from 2 to 30, with an average of 16.

Main results. Our main results are summarized in Table 2, which is structured in the same way as Table 1. The main take-away message from this experiment is that Relish can successfully solve 95% of the benchmarks within the 24 hour time limit. Among these 19 benchmarks, Relish takes an average of 94.7 seconds per benchmark, and it solves 55% of the benchmarks within 1 minute and 75% of the benchmarks within 2 minutes.

In contrast to Relish, EUSolver solves considerably fewer benchmarks within the 24 hour time-limit. In particular, Relish solves 46% more benchmarks than EUSolver (19 vs. 13) and outperforms EUSolver by 26x (in terms of running time) on the common benchmarks that can be solved by both techniques. Furthermore, similar to the previous experiment, we also observe that Relish requires fewer CEGIS iterations (7.0 vs. 9.9 on average), again confirming the hypothesis that the HFTA-based approach might have better generalization power. Finally, we note that Relish is also more efficient than EUSolver per CEGIS iteration (12 seconds vs. 492 seconds).

Memory usage. We also investigate the memory usage of Relish on the comparator benchmarks. As shown in the Mem column of Table 2, memory usage varies between 12 MB and 18367MB, with an average memory consumption of 3382MB. Comparing these statistics with Table 1, we see that memory usage is higher for comparators than the encoder/decoder benchmarks. We believe this difference can be attributed to the following three factors: First, as shown in Table 2, the size of the synthesized programs is larger for the comparator domain. Second, the relational specification for comparators is more complex and involves multiple properties such as reflexivity, anti-symmetry, totality, and transitivity. Finally, most benchmarks in the comparator domain require more CEGIS iterations to solve and therefore result in larger HFTAs.

Analysis of failed benchmarks. We manually inspected the benchmark "comparator-20" that Relish failed to synthesize within the 24 hour time limit. In particular, we found this benchmark is not expressible in our current DSL because it requires comparing integers that are obtained by concatenating all substrings that represent integers in the input strings.

Summary. In summary, this experiment demonstrates that Relish can successfully synthesize non-trivial string comparators that arise in real-world scenarios. This experiment also demonstrates the advantages of our new relational synthesis approach compared to an existing state-of-the-art solver. While the comparator synthesis task involves synthesizing a *single* function, the enumeration-based approach performs considerably worse than Relish because it does not use the relational (i.e., k-safety) specification to prune its search space.

8 RELATED WORK

In this section, we survey prior work that is most closely related to relational program synthesis.

Relational Program Verification. There is a large body of work on *verifying* relational properties about programs [Barthe et al. 2011; Benton 2004; Sousa and Dillig 2016; Sousa et al. 2018; Wang et al. 2018b; Yang 2007]. Existing work on relational verification can be generally categorized into three classes, namely *relational program logics*, *product programs*, and *Constrained Horn Clause (CHC) solving*. The first category includes Benton's Relational Hoare Logic [Benton 2004] and its variants [Barthe et al. 2012a,b; Yang 2007] as well as Cartesian Hoare Logic [Chen et al. 2017; Sousa and Dillig 2016] for verifying k-safety properties. In contrast to these approaches that provide a dedicated program logic for reasoning about relational properties, an alternative approach is to build a so-called *product program* that captures the simultaneous execution of multiple programs or different runs of the same program [Barthe et al. 2011, 2013, 2004]. In the simplest case, this approach sequentially composes different programs (or copies of the same program) [Barthe et al. 2004], but more sophisticated product construction methods perform various transformations such as loop fusion and unrolling to make invariant generation easier [Barthe et al. 2011, 2013; Sousa et al. 2014; Zaks and Pnueli 2008]. A common theme underlying all these product construction techniques is to reduce the relational verification problem to a standard safety checking problem. Another alternative approach that has been explored in prior work is to reduce the relational verification problem to solving a (recursive) set of Constrained Horn Clauses (CHC) and apply transformations that make the problem easier to solve [De Angelis et al. 2016; Mordvinov and Fedyukovich 2017]. To the best of our knowledge, this paper is the first one to address the dual *synthesis* variant of the relational verification problem.

Program Synthesis. This paper is related to a long line of work on program synthesis dating back to the 1960s [Green 1969]. Generally speaking, program synthesis techniques can be classified into two classes depending on whether they perform *deductive* or *inductive* reasoning. In particular, deductive synthesizers generate correct-by-construction programs by applying refinement and

transformation rules [Delaware et al. 2015; Kneuss et al. 2013; Manna and Waldinger 1986; Morgan 1994; Polikarpova et al. 2016]. In contrast, inductive synthesizers learn programs from input-output examples using techniques such as constraint solving [So and Oh 2017; Solar-Lezama 2009], enumerative search [Alur et al. 2017; Feser et al. 2015], version space learning [Polozov and Gulwani 2015], stochastic search [Heule et al. 2016; Schkufza et al. 2013], and statistical models and machine learning [Raychev et al. 2016, 2015]. Similar to most recent work in this area [Gulwani 2012; Polozov and Gulwani 2015; Solar-Lezama 2009; Wang et al. 2018a], our proposed method also uses inductive synthesis. However, a key difference is that we use relational examples in the form of ground formulas rather than the more standard input-output examples.

Counterexample-guided Inductive Synthesis. The method proposed in this paper follows the popular counterexample-guided inductive synthesis (CEGIS) methodology [Alur et al. 2013; Solar-Lezama 2009; Solar-Lezama et al. 2006]. In the CEGIS paradigm, an inductive synthesizer generates a candidate program P that is consistent with a set of examples, and the verifier checks the correctness of P and provides counterexamples when P does not meet the user-provided specification. Compared to other synthesis algorithms that follow the CEGIS paradigm, the key differences of our method are (a) the use of relational counterexamples and (b) a new inductive synthesis algorithm that utilizes relational specifications.

Version Space Learning. As mentioned earlier, the inductive synthesizer used in this work can be viewed as a generalization of *version space learning* to the relational setting. The notion of *version space* was originally introduced in the 1980s as a supervised learning framework [Mitchell 1982] and has found numerous applications within the field of program synthesis [Gulwani 2011; Lau et al. 2003; Polozov and Gulwani 2015; Wang et al. 2017, 2016; Yaghmazadeh et al. 2018]. Generally speaking, synthesis algorithms based on version space learning construct some sort of data structure that represents all programs that are consistent with the examples. While existing version space learning algorithms only work with input-output examples, the method proposed here works with arbitrary ground formulas representing relational counterexamples and uses hierarchical finite tree automata to compose the version spaces of individual functions.

Program Inversion. Prior work has addressed the *program inversion* problem, where the goal is to automatically generate the inverse of a given program [Chen and Udding 1990; Dijkstra 1982; Hu and D'Antoni 2017; Ross 1997; Srivastava et al. 2011]. Among these, the PINS tool uses inductive synthesis to semi-automate the inversion process based on templates provided by the user [Srivastava et al. 2011]. More recent work describes a fully automated technique, based on symbolic transducers, to generate the inverse of a given program [Hu and D'Antoni 2017]. While program inversion is one of the applications that we consider in this paper, relational synthesis is applicable to many problems beyond program inversion. Furthermore, our approach can be used to synthesize the program and its inverse *simultaneously* rather than requiring the user to provide one of these functions.

9 LIMITATIONS

While we have successfully used the proposed relational synthesis method to synthesize encoder-decoder pairs and comparators, our current approach has some limitations that we plan to address in future work. First, our method only works with simple DSLs without recursion, loops, or let bindings. That is, the programs that can be currently synthesized by Relish are compositions of built-in functions provided by the DSL. Second, we only allow relational specifications of the form $\forall\vec{x}.\phi(\vec{x})$ where ϕ is quantifier-free. Thus, our method does not handle more complex relational specifications with quantifier alternation (e.g., $\forall x.\exists y.f(x,y) = g(y,x)$).

10 CONCLUSION

In this paper, we have introduced *relational program synthesis* —the problem of synthesizing one or more programs that collectively satisfy a relational specification— and described its numerous applications in real-world programming scenarios. We have also taken a first step towards solving this problem by presenting a CEGIS-based approach with a novel inductive synthesis component. In particular, the key idea is to construct a relational version space (in the form of a hierarchical finite tree automaton) that encodes all tuples of programs that satisfy the original specification.

We have implemented this technique in a relational synthesis framework called Relish which can be instantiated in different application domains by providing a suitable domain-specific language as well as the relevant relational specifications. Our evaluation in two different application domains (namely, encoders/decoders and comparators) demonstrate that Relish can effectively synthesize pairs of closely related programs (i.e., inverses) as well as individual programs that must obey non-trivial k-safety specifications (e.g., transitivity). Our evaluation also shows that Relish significantly outperforms EUSolver, both in terms of the efficiency of inductive synthesis as well as its generalization power per CEGIS iteration.

ACKNOWLEDGMENTS

We would like to thank Yu Feng, Kostas Ferles, Jiayi Wei, Greg Anderson, and the anonymous OOPSLA'18 reviewers for their thorough and helpful comments on an earlier version of this paper. This material is based on research sponsored by NSF Awards #1712067, #1811865, #1646522, and AFRL Award #8750-14-2-0270. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proc. of FMCAD*. 1–8.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Proc. of TACAS*. 319–336.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *Proc. of FM*. Springer, 200–214.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Proc. of LFCS*. Springer, 29–43.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming* 85, 5 (2016), 847–859.

Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Proc. of CSF*. 100–114.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012a. Probabilistic relational Hoare logics for computer-aided security proofs. In *International Conference on Mathematics of Program Construction*. Springer, 1–6.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012b. Probabilistic relational reasoning for differential privacy. In *Proc. of POPL*, Vol. 47. 97–110.

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of POPL*, Vol. 39. 14–25.

Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proc. of CCS*. 875–890.

Wei Chen and Jan Tijmen Udding. 1990. Program inversion: More than fun! *Science of Comp. Prog.* 15, 1 (1990), 1–13.

Hubert Comon. 1997. Tree automata techniques and applications. *http://www. grappa. univ-lille3. fr/tata* (1997).

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2016. Relational verification through horn clause transformation. In *Proc. of SAS*. Springer, 147–169.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. of POPL*, Vol. 50. 689–700.

Edsger W Dijkstra. 1982. Program inversion. In *Selected Writings on Computing: A Personal Perspective*. Springer, 351–354.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proc. of PLDI*. 229–239.

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete applied mathematics* 42, 2-3 (1993), 177–201.

Cordell Green. 1969. Application of theorem proving to problem solving. In *Proc. of IJCAI*. 219–239.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. of POPL*. 317–330.

Sumit Gulwani. 2012. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 8–14.

Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *Proc. of PLDI*. 237–250.

Qinheping Hu and Loris D'Antoni. 2017. Automatic program inversion using symbolic transducers. In *Proc. of PLDI*. 376–389.

Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *Proc. of CAV*. 226–238.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. *Proc. of OOPSLA* 48, 10 (2013), 407–426.

Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1-2 (2003), 111–156.

Zohar Manna and Richard Waldinger. 1986. A deductive approach to program synthesis. In *Readings in artificial intelligence and software engineering*. Elsevier, 3–34.

Tom M Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.

Dmitry Mordvinov and Grigory Fedyukovich. 2017. Synchronizing constrained Horn clauses. *LPAR, EPiC Series in Computing* (2017).

Carroll Morgan. 1994. *Programming from specifications*. Prentice Hall.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proc. of ICSE*. 772–781.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proc. of PLDI*. 522–538.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. In *Proc. of OOPSLA*, Vol. 50. 107–126.

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proc. of POPL*. 761–774.

Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proc. of POPL*. 111–124.

Brian J Ross. 1997. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing* 9, 3 (1997), 331–348.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proc. of ASPLOS*. 305–316.

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Proc. of SAS*. 364–381.

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Proc. of APLAS*. Springer, 4–13.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *Proc. of ASPLOS* 41, 11 (2006), 404–415.

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proc. of PLDI*. 57–69.

Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-Way Program Merge. *PACMPL* OOPSLA (2018).

Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. 2014. Consolidation of queries with user-defined functions. In *Proc. of PLDI*, Vol. 49. 554–564.

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S Foster. 2011. Path-based inductive synthesis for program inversion. In *Proc. of PLDI*, Vol. 46. 492–503.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *PACMPL* 1, OOPSLA (2017), 62:1–62:26.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018a. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.

Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: filtering spreadsheet data using examples. In *Proc. of OOPSLA*. 195–213.

Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018b. Verifying equivalence of database-driven applications. *PACMPL* 2, POPL (2018), 56:1–56:29.

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018c. Relational Program Synthesis. http://arxiv.org/abs/1809.02283. arXiv:1809.02283

Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables using Programming-by-Example. *Proc. of VLDB* 11, 5 (2018), 580–593.

Hongseok Yang. 2007. Relational separation logic. *Theoretical Computer Science* 375, 1-3 (2007), 308–334.

Anna Zaks and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In *Proc. of FM*. Springer, 35–51.