Functional Synthesis with Examples

Grigory Fedyukovich and Aarti Gupta

Princeton University, Princeton, USA, {grigoryf,aartig}@cs.princeton.edu

Abstract. Functional synthesis (FS) aims at generating an implementation from a declarative specification over sets of designated input and output variables. Traditionally, FS tasks are formulated as $\forall\exists$-formulas, where input variables are universally quantified and output variables are existentially quantified. State-of-the-art approaches to FS proceed by eliminating existential quantifiers and extracting Skolem functions, which are then turned into implementations. Related applications benefit from having concise (i.e., compact and comprehensive) Skolem functions. In this paper, we present an approach for extracting concise Skolem functions for FS tasks specified as *examples*, i.e., tuples of concrete values of integer variables. Our approach builds a decision tree from *relationships* between inputs and outputs and *preconditions* that classify all examples into subsets that share the same input-output relationship. We also present an extension that is applied to *hybrid* FS tasks, which are formulated in part by examples and in part by arbitrary declarative specifications. Our approach is implemented on top of a functional synthesizer AE-VAL and evaluated on a set of reactive synthesis benchmarks enhanced with examples. Solutions produced by our tool are an order of magnitude smaller than ones produced by the baseline AE-VAL.

1 Introduction

One way to ensure the absence of bugs in programs is to replace a human developer with a machine that leverages automated decision procedures and theorem provers to develop programs that are correct-by-construction. But the task of automatically synthesizing programs from given specifications is notoriously hard and often depends crucially on the way specifications are formulated. Furthermore, it is often tedious to formulate a specification precisely and completely, such that it adequately represents the targeted intent. For humans, it is usually easier to provide a set of examples, such as tuples of concrete values for input and output variables. The task of the automated synthesizer is to generate an implementation, which produces given outputs for given inputs. In addition, the synthesizers should envision as much as possible which input-output tuples could appear in the actual programs, and implementations should be general enough to cover such cases. Finally, a specification may also include arbitrary additional requirements, and the resulting implementation should be consistent with both input-output examples and constraints at the same time.

Many different techniques have been studied under the general umbrella of program synthesis, with a wide range in kinds of specifications and search techniques [1–3,26–29,31]. Typically, Functional Synthesis (FS) requires a declarative *relational specification* which connects input and output variables. Programming by Examples (PBE) requires a set of input-output tuples consisting of *concrete values* of variables. Although both FS and PBE have been developed successfully in many domains, there is a relative lack of unifying efforts that take advantages of them together to provide a general solution.

A classic formulation of an FS task is via checking the validity of a $\forall\exists$-formula, in which inputs are universally quantified, and outputs are existentially quantified. The validity of this formula guarantees realizability of the synthesis task, and a witnessing Skolem function can be turned into an implementation. In [11], Skolem functions are generated while lazily eliminating quantifiers in (and proving the validity of) $\forall\exists$-formulas in linear integer arithmetic (LIA). The generated solutions are represented in the form of decision trees, where decision nodes denote formulas over inputs called *preconditions*, and leaves denote equalities of outputs with terms over inputs called *local Skolem terms*. This method can be applied in a straightforward manner to a PBE task too: in the corresponding decision tree, the preconditions would be represented by (conjunctions of) equalities over inputs and their values, and the local Skolem terms would simply be the corresponding values of the output variables.

To obtain concise Skolem terms, decision trees can be compacted by potentially merging decision nodes that could share the same leaves. In the context of PBE, this idea is in general inapplicable because the terms in the leaves are always constants. To apply any compaction, there should be a way to replace these constants by terms over inputs. For LIA, this can be done by discovering linear equations over input and output variables.

The challenge is that not all given examples would be classified by a single linear equation. Thus, in our approach, we first *partition* the set of examples into subsets, such that all examples within each subset share the same linear relationship. Clearly, such a partitioning is not unique, and we target deriving a small number of subsets. Another criterion we consider is that all examples within each subset should be classified concisely by some precondition over inputs. In particular, a precondition that simply disjoins all equalities between inputs and concrete values would be too bulky (growing linearly with the size of the subset). Instead, we seek an opportunity to replace it by some inequality or conjunction of inequalities. These criteria lead to compact decision trees.

One key novelty in our approach is a completely automated procedure to discover compact preconditions and local Skolem terms for PBE tasks in LIA. Existing synthesis approaches, e.g., those based on enumerative search [26, 27, 31], require the user to additionally supply formal grammars (or templates) that specify a pool of candidate formulas and terms. They search for suitable candidates from the grammars and iteratively test them on given examples. While this general capability is very useful for rich grammars and specifications, for LIA it is possible to completely automate these steps. In particular, our approach does not require any extra input from the user and automatically *infers* candidates for local Skolem terms directly from data using canonical equations in linear arithmetic.

The candidate preconditions are also inferred automatically from data – they specify ranges of input values. To find suitable candidates, we pose queries over certain ranges in the ∀∃-form which intuitively say "for all inputs within the candidate range, there exists an output value which is consistent with the candidate assignment and all given examples". By counting how many examples are covered by each candidate that passes the ∀∃-test, we create a ranking of candidates and pick those with the highest rank.

We also extended the approach to *hybrid* PBE and FS tasks, formulated in part by using input-output examples and in part by using arbitrary input-output relational constraints. To solve such problems, the formula describing an FS-part of the task is simply added to our ∀∃-test and is taken into account when filtering suitable candidate ranges and the corresponding local Skolem terms.

Our implementation on top of the AE-VAL [11] tool has been evaluated on a range of reactive synthesis benchmarks enhanced with examples. The discovered solutions are an order of magnitude smaller than straightforward Skolem terms and less sensitive to the number of examples.

2 Running Example

Table 1 gives a set of examples by means of integer values of input variables x_1, x_2, and x_3, and an output variable y. Each row represents a transition from concrete inputs to the concrete output. Some examples are *incomplete*, i.e., a subset of input values is not given (denoted "$\cdot$"). For instance, the first row specifies input values for only x_1 and x_3; and it should be interpreted as "if both x_1 and x_3 are equal to one, then y should be equal to one as well".

Table 1: Input-output tuples.

x_1	x_2	x_3	y
1	$\cdot$	1	1
0	$\cdot$	2	0
$\cdot$	1	3	2
$\cdot$	2	4	4
2	4	5	6
2	0	6	2

Our goal is to find 1) symbolic linear relationships among given values in each input-output tuple, and 2) preconditions that uniquely determine equivalence classes of these relationships. For instance, for the first two rows, it is true that $y = x_1$. Precondition $1 \le x_3 \le 2$ uniquely determines the first two rows, in a sense that for the remaining four rows, it does not hold. Similarly, for the next two rows, $y = 2 \cdot x_2$ under precondition $3 \le x_3 \le 4$; and for the last two rows, $y = x_1 + x_2$ under precondition $5 \le x_3 \le 6$. Combining preconditions and relationships, we can formally describe how y can be computed from x_1, x_2, and x_3:

$$y = ite(1 \le x_3 \le 2, x_1, ite(3 \le x_3 \le 4, 2 \cdot x_2, x_1 + x_2))$$

In fact, such a *decision tree* is not unique for values in the table. A more compact one can be found by our algorithm:

$$y = ite(2 \le x_3 \le 5, 2 \cdot x_3 - 4, x_1)$$

In the rest of the paper, we show how such a solution can be discovered automatically.

3 Background and Notation

A many-sorted first-order theory consists of disjoint sets of sorts $\mathcal{S}$, function symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. A set of *terms* is defined recursively as follows:

$$\mathit{term} ::= f(\mathit{term}, \ldots, \mathit{term}) \mid \mathit{const} \mid \mathit{var}$$

where $f \in \mathcal{F}$, *const* is an application of some $v \in \mathcal{F}$ of zero arity, and *var* is a variable uniquely associated with a sort in $\mathcal{S}$. A set of quantifier-free *formulas* is built recursively using the usual grammar:

$$\begin{aligned}\mathit{formula} ::= & \, \mathit{true} \mid \mathit{false} \mid p(\mathit{term}, \ldots, \mathit{term}) \mid \mathit{Bvar} \mid \\ & \neg \mathit{formula} \mid \mathit{formula} \wedge \mathit{formula} \mid \mathit{formula} \vee \mathit{formula}\end{aligned}$$

where *true* and *false* are Boolean constants, $p \in \mathcal{P}$, and *Bvar* is a variable associated with sort *Bool*.

In this paper, we consider the theory Linear Integer Arithmetic (LIA). In LIA, $\mathcal{C} \overset{\text{def}}{=} \{\mathbb{Z}, \mathit{Bool}\}$, $\mathcal{F} \overset{\text{def}}{=} \{+, \cdot, \mathit{div}\}$, and $\mathcal{P} \overset{\text{def}}{=} \{=, >, <, \geq, \leq, \neq\}$. We define *ite* as a shortcut for *if-then-else*, i.e., $\mathit{ite}(x, y, z) \overset{\text{def}}{=} (x \wedge y) \vee (\neg x \wedge z)$.

Formula φ is called satisfiable if there exists an interpretation m, called a model, of each element (i.e., a variable, a function or a predicate symbol), under which φ evaluates to *true*; otherwise φ is called unsatisfiable. If every model of φ is also a model of ψ, then we write $\varphi \implies \psi$. A formula φ is called *valid* if $\mathit{true} \implies \varphi$.

For existentially-quantified formulas of the form $\exists y \,.\, \psi(\vec{x}, y)$, validity requires that each interpretation for variables in $\vec{x}$ and each function and predicate symbol in ψ can be *extended* to a model of $\psi(\vec{x}, y)$. For a valid formula $\exists y \,.\, \psi(\vec{x}, y)$, a term $\mathit{sk}_y(\vec{x})$ is called a *Skolem term*, if $\psi(\vec{x}, \mathit{sk}_y(\vec{x}))$ is valid.

In the paper, we assume that all free variables $\vec{x}$ are implicitly universally quantified. For simplicity, we omit the arguments and simply write φ when the arguments are clear from the context.

Extracting Skolem terms. Our work is built on top of a lazy quantifier-elimination method for checking validity and performing synthesis called AE-VAL [11,12]. It generates a structured synthesis solution in the form of a decision tree. Its main procedure is based on deriving a sequence of Model-Based Projections (MBPs) [19] to lazily decompose the overall problem, where each model is used to derive a precondition that captures an arbitrary subspace on the $\vec{x}$ variables and a Skolem term for the $\vec{y}$ variables. Unlike other prior work [21], AE-VAL does not require converting the formula into Disjunctive Normal Form (DNF), which often leads to larger and redundant solutions. AE-VAL also uses minimization and compaction procedures for on-the-fly compaction of the generated synthesis solution. In particular, it derives Skolem terms that can be re-used across multiple preconditions for a single output and shares the preconditions in a common decision tree across multiple outputs in a program. This is done by identifying theory terms that can be shared both within and across outputs.

However, AE-VAL handles relational specifications (FS tasks) only, and it is not designed to handle input-output examples (PBE tasks) properly. When given concrete input-output examples, it would generate an implementation in the form of a decision tree with the depth equal to the number of examples (as described in more detail in the next section).

4 Synthesis by Examples

We formalize the case when all examples are complete and defer the case of partially defined examples till the next section.

Definition 1. *Let $\vec{x} = \langle x_1, \ldots, x_n \rangle$ be a vector of input variables and E be a set of m examples, where each $\vec{e} \in E$ is a vector of integers and $\vec{e}$ has $n+1$ components. For an output variable y, vectors $\vec{x}$ and $\vec{e} \in E$ are connected through an* example-formula *ζ:*

$$\zeta(e, \vec{x}, y) \overset{\text{def}}{=} \bigwedge_{1 \le i \le n} (\vec{x}[i] = \vec{e}[i]) \implies y = \vec{e}[n+1]$$

We assume consistency among all examples in E, i.e., that the following formula is valid:

$$\forall \vec{x} \,.\, \exists y \,.\, \bigwedge_{\vec{e} \in E} \zeta(\vec{e}, \vec{x}, y) \tag{1}$$

Note that the formula could only be invalid if there are two vectors $\vec{e}_1, \vec{e}_2 \in E$, such that:

$$\vec{e}_1[n+1] \neq \vec{e}_2[n+1] \wedge \forall i \,.\, 0 \le i \le n \implies \vec{e}_1[i] = \vec{e}_2[i]$$

A Skolem term for y in (1) can be derived in the form of a nested *ite*-block of depth m as shown below:

$$ite\Big(\bigwedge_{1 \le i \le n} (\vec{x}[i] = \vec{e}_1[i]), \vec{e}_1[n+1], ite\big(\bigwedge_{1 \le i \le n} (\vec{x}[i] = \vec{e}_2[i]), \vec{e}_2[n+1], \ldots, 0\big)\Big) \tag{2}$$

where each $\vec{e}_i \in E$ identifies the i-th level of the decision tree, and the last *else*-branch represents the case when none of examples match current values of $\vec{x}$, thus an arbitrary value (e.g., 0 as in (2)) can be assigned to y.

We wish to generate a Skolem term for y as a decision tree with a smaller depth. That is, among the space of terms of form (3), we wish to identify the one with a (preferably) minimal number of *ite*-blocks.

Definition 2. *A Skolem term for an example-formula* (1) *is called* generalized *if it has the following form:*

$$ite\Big(pre[1](\vec{x}), sk[1](\vec{x}), ite\big(pre[2](\vec{x}), sk[2](\vec{x}), \ldots, 0\big)\Big) \tag{3}$$

where the vector pre collects formulas over $\vec{x}$ (called preconditions*), and the vector sk collects terms over $\vec{x}$ (called* local Skolems*). Each pair $\langle pre_F, sk_F \rangle$ corresponds to a subset of examples $F \subseteq E$, such that* (4) *and* (5) *hold:*

$$\forall \vec{e} \in F \,.\, \bigwedge_{1 \le i \le n} (\vec{x}[i] = \vec{e}[i]) \implies pre_F(\vec{x}) \tag{4}$$

$$\forall \vec{e} \in F \,.\, \bigwedge_{1 \le i \le n} (\vec{x}[i] = \vec{e}[i]) \wedge y = sk_F(\vec{x}) \implies y = \vec{e}[n+1] \tag{5}$$

We present an algorithm that partitions the given set E into disjoint subsets, which give rise to vectors *pre* and *sk*. An overview of the proposed algorithm is shown in Alg. 1. The key insight is to identify each subset $F \subseteq E$ by inferring a precondition pre_F and a local Skolem term sk_F from pairs of examples $\langle \vec{e}_1, \vec{e}_2 \rangle \in E \times E$. The algorithm relies on helper procedures to discover a candidate precondition (line 4) and a candidate term for each pair (line 5). These procedures, applied to all pairs of examples, produce a set of candidate preconditions and a set of candidate terms. However, there is no guarantee that a precondition and a term, which suit all given examples, could be discovered. But we can often find some precondition and some term that will suit *many* examples, which will constitute the desired subset F. In order to identify it, our algorithm filters bad preconditions and terms and ranks successful ones. In the rest of this section, we outline a particular instantiation of subroutines of Alg. 1 for LIA[1].

Method GETRANGE. To define a range of values of variables $\vec{x}$ between $\vec{e}_1$ and $\vec{e}_2$ we introduce a function M:

$$M(\vec{e}_1, \vec{e}_2, i) \overset{\text{def}}{=} \begin{cases} \vec{e}_1[i] \leq \vec{x}[i] \wedge \vec{x}[i] \leq \vec{e}_2[i], & \text{if } \vec{e}_1[i] \leq \vec{e}_2[i] \\ \vec{e}_2[i] \leq \vec{x}[i] \wedge \vec{x}[i] \leq \vec{e}_1[i], & \text{otherwise} \end{cases}$$

Then, formula γ representing a range between $\vec{e}_1$ and $\vec{e}_2$ is simply computed as:

$$\gamma \overset{\text{def}}{=} \bigwedge_{1 \leq i \leq n} M(\vec{e}_1, \vec{e}_2, i) \tag{6}$$

Method CONNECT. Relationships between variables $\vec{x}$ and y are determined by a *canonical equation of a line* and two vectors of their values, $\vec{e}_1$ and $\vec{e}_2$:

$$\frac{\vec{x}[1] - \vec{e}_1[1]}{\vec{e}_2[1] - \vec{e}_1[1]} = \ldots = \frac{\vec{x}[n] - \vec{e}_1[n]}{\vec{e}_2[n] - \vec{e}_1[n]} = \frac{y - \vec{e}_1[n+1]}{\vec{e}_2[n+1] - \vec{e}_1[n+1]} \tag{7}$$

It gives rise to various possible equalities connecting components of $\vec{x}$ and y. In particular, any two equalities of form $(\vec{x}[i] - \vec{e}_1[i]) \cdot (\vec{e}_2[n+1] - \vec{e}_1[n+1]) = (\vec{e}_2[i] - \vec{e}_1[i]) \cdot (y - \vec{e}_1[n+1])$, where $1 \leq i \leq n$, can be summed (or subtracted) side-by-side.

Example 1. Recall our set of input-output tuples from Sect. 2. Suppose, in the first loop of Alg. 1, we are considering the first two tuples (i.e., rows in Table 1): $\zeta_1 \overset{\text{def}}{=} (x_1 = 1 \wedge x_3 = 1) \implies (y = 1)$ and $\zeta_2 \overset{\text{def}}{=} (x_1 = 0 \wedge x_3 = 2) \implies (y = 0)$. The GETRANGE method produces $\gamma_{1,2} \overset{\text{def}}{=} 0 \leq x_1 \leq 1 \wedge 1 \leq x_3 \leq 2$. The CONNECT method produces equalities $X_{1,2} = \{y = x_1,\ y = 2 - x_3$, and $2 \cdot y = x_1 - x_3 + 2\}$ (the last one is produced by summing left and right sides of the first two equalities).

[1] With the required support for quantifier elimination, it can be immediately be adapted to rational arithmetic, nonlinear arithmetic, and bitvectors. But to achieve more compact solutions, these algorithms could benefit from additional adjustments in method CONNECT which are left for future work.

Algorithm 1: $\textsc{getBestClass}(\vec{x}, y, E)$

```
Input: $\vec{x}, y, E$
Output: $F, pre_F, sk_F$, s.t. (4) and (5) hold
$Cands \leftarrow \varnothing$;
$R \leftarrow \lambda\xi \,.\, \varnothing$;
for $\langle \vec{e}_1, \vec{e}_2 \rangle \in E \times E$ do
    $\gamma \leftarrow \textsc{getRange}(\vec{e}_1, \vec{e}_2)$;
    $X \leftarrow \textsc{connect}(\vec{e}_1, \vec{e}_2)$;
    for $\xi \in X$ do
        if $\textsc{sanityTest}(\gamma, \xi)$ then
            $\xi \leftarrow \textsc{localSkolem}(\gamma, \xi)$;
            $Cands \leftarrow Cands \cup \{\xi\}$;
            $R(\xi) \leftarrow R(\xi) \cup \{\gamma\}$;
for $\xi \in Cands$ do
    $R(\xi) \leftarrow R(\xi) \cup \textsc{generalize}(R(\xi))$;
    for $\gamma \in R(\xi)$ do $\textsc{rank}(\gamma, \xi, E)$;
return $\textsc{largest}(E, \gamma, \xi)$;
```

Algorithm 2: $\mathrm{PBE}(\vec{x}, y, E)$

```
Input: $\vec{x}, y, E$
Output: Skolem term $sk$ for $y$ in (1)
if $E = \varnothing$ then return $\textsc{pickAny}(\mathbb{Z})$;
$F, pre_F, sk_F \leftarrow \textsc{getBestClass}(\vec{x}, y, E)$;
if $F = \varnothing$ then
    $\vec{e} \leftarrow \textsc{pickAny}(E)$;
    $F \leftarrow \{\vec{e}\}$;
    $pre_F \leftarrow \bigwedge_{1 \le i \le n} (\vec{x}[i] = \vec{e}[i])$;
    $sk_F \leftarrow (y = \vec{e}[n+1])$;
return $ite(pre_F, sk_F, \mathrm{PBE}(\vec{x}, y, E \setminus F))$;
```

Methods $\textsc{sanityTest}$ *and* $\textsc{localSkolem}$. Let γ be a range-formula over $\vec{x}$, and ξ be a formula over $\vec{x}$ and y. We filter a set of pairs $\langle \gamma, \xi \rangle$ based on the following criterion:

$$\forall \vec{x} \,.\, \gamma(\vec{x}) \implies \exists y \,.\, \xi(\vec{x}, y) \tag{8}$$

If formula (8) is valid, a Skolem term for y exists and can be extracted, e.g., using the AE-VAL algorithm.

Example 2. Recall ζ_1 and ζ_2 produced from our input-output tuples (see Sect. 2) in Example 1. For each $\xi \in X_{1,2}$ and $\gamma_{1,2}$, we pose a query of form (8):

$$\forall x_1, x_2, x_3 \,.\, 0 \leq x_1 \leq 1 \wedge 1 \leq x_3 \leq 2 \implies \exists y \,.\, y = x_1 \qquad \text{valid}$$
$$\forall x_1, x_2, x_3 \,.\, 0 \leq x_1 \leq 1 \wedge 1 \leq x_3 \leq 2 \implies \exists y \,.\, y = 2 - x_3 \qquad \text{valid}$$
$$\forall x_1, x_2, x_3 \,.\, 0 \leq x_1 \leq 1 \wedge 1 \leq x_3 \leq 2 \implies \exists y \,.\, 2 \cdot y = x_1 - x_3 + 2 \qquad \text{invalid}$$

The results for these queries are shown on the right. Since the last query is invalid, we thus proceed with the other candidates $y = x_1$ and $y = 2 - x_3$ only.

Note that if for some $\xi \in X$, the coefficient for y is 1, then any query of form (8) is valid (and a Skolem function for y is ξ itself).

Method GENERALIZE. Given a set of factored preconditions of ξ (recall (6)), we fix a variable $x \in \vec{x}$ and identify factors over x across all preconditions. Then, we iteratively prune this set of formulas by applying the following rule:

$$\frac{\alpha_1 \leq x \wedge x \leq \alpha_2 \quad \alpha_3 \leq x \wedge x \leq \alpha_4}{min(\alpha_1, \alpha_3) \leq x \leq max(\alpha_2, \alpha_4)} \text{ if } \alpha_3 \leq \alpha_2 \wedge \alpha_1 \leq \alpha_4$$

Repeating this operation yields a new formula over x. Repeating this for all $x \in \vec{x}$ and conjoining the resulting formulas gives us a new range-formula for ξ.

Note that this new formula is an over-approximation of the disjunction of the original preconditions for ξ. By using these preconditions for all candidate Skolem terms, we face a trade-off between the depth of the resulting decision tree and the syntactic size of preconditions. That is, some of the over-approximated preconditions could be too coarse, and thus filtered away (see method RANK of the algorithm). But if an over-approximated precondition has not been filtered, it is likely to be more compact and general.

Example 3. Let $\gamma_{1,6} \overset{\text{def}}{=} 1 \leq x_1 \leq 2 \wedge 1 \leq x_3 \leq 6$ and $X_{1,6} = \{y = x_1\}$.[2] Following Examples 1 and 2, $y = x_1$ is also associated with $\gamma_{1,2} = 0 \leq x_1 \leq 1 \wedge 1 \leq x_3 \leq 2$. Thus, our generalization produces $\gamma_{1,2,6} \overset{\text{def}}{=} 0 \leq x_1 \leq 2 \wedge 1 \leq x_3 \leq 6$.

Methods RANK *and* LARGEST. These two methods identify the best formula (or a combination of formulas) among the candidates. We evaluate a precondition γ and a suitable candidate local Skolem term ξ on all examples $\vec{e} \in E$. In particular, we identify a subset of examples, for which implication (9) holds (denoted $F(\gamma)$) and a subset of examples, for which implication (10) does not hold (denoted $G(\xi)$).

$$\bigwedge_{1 \leq i \leq n} (\vec{x}[i] = \vec{e}[i]) \implies \gamma(\vec{x}) \tag{9}$$

$$\xi(\vec{x}) \wedge \bigwedge_{1 \leq i \leq n} (\vec{x}[i] = \vec{e}[i]) \implies y = \vec{e}[n+1] \tag{10}$$

[2] We refer the reader to Sect. 5 that describes a process of learning from partial examples.

Cardinalities of $F(\gamma)$ and $G(\xi)$ give a ranking to each $\langle \gamma, \xi \rangle$. If $G(\xi)$ is nonempty, then the ranking is zero. Otherwise, the ranking is $|F(\gamma)|$.

Example 4. To rank precondition $\gamma_{1,2,6}$ for a candidate $y = x_1$ generated in Example 3, we enumerate all input-output tuples from Table 1 and test implications (9) and (10). It appears that set $G(y = x_1)$ is nonempty since for the fifth tuple (10) is invalid:

$$y = x_1 \wedge x_1 = 2 \wedge x_2 = 4 \wedge x_3 = 5 \not\Longrightarrow y = 6$$

Another precondition $\gamma_{2,3,4,5} \overset{\text{def}}{=} 2 \leq x_3 \leq 5$ for candidate $y = 2 \cdot x_3 - 4$ (computed similarly) gets ranking 4 since set $G(y = 2 \cdot x_3 - 4)$ is empty, and $F(\gamma_{2,3,4,5})$ consists of four examples.

Since ranking explicitly checks partially generated functions w.r.t. specifications, our solutions are correct by construction. More formally, it is represented by the following lemma.

Lemma 1. *Any pair of formulas $\langle \gamma, \xi \rangle$ with a non-zero ranking can be used to extract the outer ite-block of the Skolem term.*

For getting a candidate formula with the best coverage, we select the formula with a higher (and non-zero) ranking. It intuitively corresponds to the largest subset of examples that can be described by a single precondition and a single local Skolem term.

Alg. 2 describes an algorithm to construct a decision tree recursively. It starts with the full set of given examples E, and uses Alg. 1 to identify the largest subset $F \subseteq E$, elements of which share the same precondition and local Skolem term (to be used at one level of the decision tree). In the case when F is empty, it is enough to pick any element of E and create a precondition and a local Skolem term in a straightforward way. Then, all elements of F are excluded from E, and the algorithm recurses. It converges when E is empty, and for this (the deepest) level of the decision tree, we can pick any local Skolem term (e.g., an integer constant) with no precondition.

5 Synthesis by Partial Examples

In this section, we present a generalization of the synthesis by examples algorithm (described in Sect. 4) that relies on *subvectors* of examples.

Definition 3. *Let $\vec{x}$ be a vector containing n components and s be an injective function to $\{1, \ldots, n\}$. A* subvector *of $\vec{x}$ (denoted $\vec{x}_{|s}$) is a vector, such that for all i, $\vec{x}_{|s}[i] = \vec{x}[s(i)]$.*

Intuitively, $\vec{x}_{|s}$ is produced by *removing* components from $\vec{x}$ and preserving the order of the remaining components. We naturally extend this definition to sets of vectors, i.e., $E_{|s} \overset{\text{def}}{=} \{\vec{e}_{|s} \mid \vec{e} \in E\}$.

The algorithms from Sect. 4 can be used for subvectors of input variables and sets of subvectors of examples. In particular, let s be an injective function to $\{1, \ldots, n\}$, we can apply Alg. 1 to $\vec{x}_{|s}$ and $E_{|s}$, if the following formula is valid:

$$\forall \vec{x}_{|s} \,.\, \exists y \,.\, \bigwedge_{\vec{e} \in E} \zeta(\vec{e}_{|s}, \vec{x}_{|s}, y) \tag{11}$$

There are two main advantages for doing this. First, it may give us more concise and general solutions (which are expressible using fewer variables). Second, while extracting subvectors, we shrink the set of examples, which lowers the cost of the synthesis procedure.

Thus, the whole procedure can be supplied with a preprocessing, during which various mappings s are considered and formulas of the form (11) are checked for validity. The mapping s with the smallest domain size can be then used for synthesis by examples. The speed of the entire procedure could then be improved, but the effectiveness of the resulting solution could worsen.

Example 5. Recall Example 1, let s be a function with $dom(s) = \{1\}$ and $img(s) = \{3\}$. Then $E_{|s}$ is constructed from E by keeping the values of x_3. The formula (11) is compiled as follows:

$$\begin{aligned} \forall x_3, \exists y \,.\, &(x_3 = 1 \implies y = 1) \wedge (x_3 = 2 \implies y = 0) \wedge (x_3 = 3 \implies y = 2) \\ &(x_3 = 4 \implies y = 4) \wedge (x_3 = 5 \implies y = 6) \wedge (x_3 = 6 \implies y = 2) \end{aligned}$$

The formula above is valid, and Alg. 2 can be applied to extract the following Skolem term:

$$y = ite(2 \leq x_3 \leq 5, 2 \cdot x_3 - 4, ite(x_3 = 1, 1, 2))$$

Note that this Skolem term is not optimal (the one provided in Sect. 2 has a fewer nested *ite*-blocks). A heuristic in the rest of the section aims at discovering a more effective solution.

Some of examples could be defined only partially, i.e., using a sequence of injective functions $s_1, \ldots, s_m$ to $\{1, \ldots, n\}$ that gives rise to sequences $\vec{x}_{|s_1}, \ldots, \vec{x}_{|s_m}$ and $E_1, \ldots, E_m$. For each s_i, examples from E_i use values of $\vec{x}_{|s_i}$ and y.

The task is to extract a Skolem term for the given *valid* formula:

$$\forall \vec{x} \,.\, \exists y \,.\, \bigwedge_{1 \leq i \leq m} \; \bigwedge_{\vec{e} \in E_i} \zeta(\vec{e}, \vec{x}_{|s_i}, y) \tag{12}$$

Alg. 3 shows an adaptation of Alg. 2 applicable to the union of sets of all examples $E \overset{\text{def}}{=} E_1 \cup \ldots \cup E_m$. It iteratively produces subvectors of all examples and finds such a subset of them, which gives the valid example-formula (line 3). Then, it applies Alg. 1 to detect a level of the decision tree (line 4) and shrinks the set of examples accordingly (lines 10-13). Similarly to Alg. 2, the algorithm recurses until the entire decision tree is constructed (line 3).

Theorem 1. *If* $\bigcap_{1 \leq i \leq m} img(s_i) \neq \varnothing$*, then Alg. 3 returns a Skolem term for* (12).

Example 6. In the first iteration, Alg. 3 considers function s from Example 5. As a result, it extracts four input-output tuples (recall Example 4). In the second

```
Algorithm 3: partialExsPBE($\vec{x}, y, \{\langle s_i, E_i\rangle\}_{1\leq i\leq n}$)
   Input: $\vec{x}, y$: variables, $\{\langle s_i, E_i\rangle\}_{1\leq i\leq n}$: set of pairs of
          functions and sets of partial examples, $E = \bigcup_{1\leq i\leq n} E_i$
   Output: Skolem term $sk$ for $y$ in (12)
 1 if $E = \varnothing$ then return pickAny($\mathbb{Z}$);
 2 let $s$ be such that $\forall s_j, img(s) \subseteq img(s_j)$;
 3 $E' \leftarrow$ getValidSubset($E_{|s}$);
 4 $F, pre_F, sk_F \leftarrow$ getBestClass($\vec{x}_{|s}, y, E'$);
 5 if $F = \varnothing$ then
 6     $\vec{e} \leftarrow$ pickAny($E'$);
 7     $F \leftarrow \{\vec{e}\}$;
 8     $pre_F \leftarrow \bigwedge_{i\in img(s)} (\vec{x}[i] = \vec{e}[i])$;
 9     $sk_F \leftarrow (y = \vec{e}[n+1])$;
10 $Rem \leftarrow \varnothing$;
11 for $1 \leq i \leq n$ do
12     $E_i \leftarrow \{e \in E_i \mid \vec{e}_{|s} \notin E'\}$;
13     if $E_i \neq \varnothing$ then $Rem \leftarrow Rem \cup \langle s_i, E_i\rangle$;
14 return $ite(pre_F, sk_F,$ partialExsPBE($\vec{x}, y, Rem$));
```

iteration, Alg. 3 takes as input just two remaining tuples and considers function s', such that $dom(s') = \{1\}$ and $img(s') = \{1\}$. It appears that $\gamma_{1,6}$ and the $y = x_1$ are considered again (recall Example 1). But in this case (as opposed to Example 4), their ranking is computed with respect to only two input-output tuples, thus resulting in $F(\gamma_{1,6}) = \varnothing$ and $|G(y = x_1)| = 2$. This concludes the search, and the final Skolem term gets composed from two nested *ite*-blocks (i.e., exactly as provided in Sect. 2).

6 Hybrid Synthesis: PBE + FS

Suppose we are given an additional requirement $\psi(\vec{x}, y)$ for the input and output variables. Note that ψ may be a *partial* specification, i.e., it may impose necessary but not sufficient conditions for correctness. The goal is to discover a Skolem term for (13):

$$\forall \vec{x} \,.\, \exists y \,.\, \bigwedge_{\vec{e}\in E} \zeta(\vec{e}, \vec{x}, y) \wedge \psi(\vec{x}, y) \tag{13}$$

If ψ is consistent with the set of examples, then a Skolem term for (13) can be discovered by the procedure from Sect. 4 with the following differences:

– *Default local Skolem term in Alg. 2 (line 1).*
  The random choice is replaced with a Skolem term for y in formula $\forall \vec{x} \,.\, \exists y \,.\, \psi(\vec{x}, y)$ (i.e., solve a standard functional synthesis task without examples). In our implementation, we use AE-VAL.

– *Criteria* (8) *and* (10) *for methods* SANITYTEST *and* RANK*, respectively.* We check the validity of formulas, respectively (14) and (15), enhanced with ψ.

$$\forall \vec{x} \,.\, \gamma(\vec{x}) \implies \exists y \,.\, \xi(\vec{x}, y) \wedge \psi(\vec{x}, y) \tag{14}$$

$$\xi(\vec{x}) \wedge \bigwedge_{1 \leq i \leq n} (\vec{x}[i] = \vec{e}[i]) \implies y = \vec{e}[n+1] \wedge \psi(\vec{x}, y) \tag{15}$$

– *Extra criterion for method* GENERALIZE. We perform an extra sanity check (14) for each over-approximated precondition.

Theorem 2. *With these adjustments, the output of Alg. 2 is a Skolem term for* (13)*.*

Example 7. Consider a synthesis task consisting of 1) values specified in Table 1, and 2) an additional requirement $y \geq x_1 \vee y \geq x_2$. Thus, the entire formula is as follows:

$$\begin{aligned}
&\forall x_1, x_2, x_3 \,.\, \exists y \,.\, y \geq x_1 \vee y \geq x_2 \wedge \\
&\qquad (x_1 = 1 \wedge x_3 = 1 \implies y = 1) \wedge (x_1 = 0 \wedge x_3 = 2 \implies y = 0) \wedge \\
&\qquad (x_2 = 1 \wedge x_3 = 3 \implies y = 2) \wedge (x_2 = 2 \wedge x_3 = 4 \implies y = 4) \wedge \\
&(x_1 = 2 \wedge x_2 = 4 \wedge x_3 = 5 \implies y = 6) \wedge (x_1 = 2 \wedge x_2 = 0 \wedge x_3 = 6 \implies y = 2)
\end{aligned}$$

This is a suitable task for AE-VAL, but it would return a Skolem term as a decision tree with six levels. In contrast, our algorithm produces a Skolem term for y with just three levels:

$$\begin{aligned}
&\mathit{ite}\Big(4 \leq x_3 \leq 6 \wedge 0 \leq x_2 \leq 4,\, x_2 + 2, \\
&\qquad\qquad \mathit{ite}\big(1 \leq x_3 \leq 2 \wedge 0 \leq x_1 \leq 1, x_1, \mathit{ite}(x_2 = 1 \wedge x_3 = 3, 2, x_1)\big)\Big)
\end{aligned}$$

The deepest decision, x_1, is a Skolem term for y in formula $\forall x_1, x_2, x_3 \,.\, \exists y \,.\, y \geq x_1 \vee y \geq x_2$. The preconditions and relationships identified in Example 6 are not suitable.

7 Evaluation

We implemented our synthesis algorithm on top of the AE-VAL [11] tool[3] which uses the Z3 SMT solver [7]. To compare our implementation with state-of-the-art tools, we considered the "plain" AE-VAL, CVC4 [23], EUSOLVER [3] and DRYADSYNTH [15]. None of them supports discovery of relationships among data tuples: AE-VAL, CVC4, and DRYADSYNTH return a *straightforward Skolem*, i.e., a formula of form (2) with nested *ite*-blocks of the highest depth m; and EUSOLVER has frontend issues. The timings for discovery of a straightforward Skolem are usually small even for a large number of examples. Since we do not consider a straightforward Skolem an acceptable solution for our class of tasks,

[3] The source code and benchmarks are available at `https://github.com/grigoryfedyukovich/aeval`.

we do not present a detailed evaluation report for the competing tools. Instead, we focus on details of our AE-VAL-PBE and AE-VAL.

We considered 59 benchmarks from various Assume-Guarantee contracts written in the Lustre programming language [17]. These are the relational specifications derived mainly from industrial projects, such as a Quad-Redundant Flight Control System, a Microwave model, a Generic Patient Controlled Analgesia infusion pump, a Cinderella-Stepmother game, and several tricky hand-written examples. The depths of solutions for these original benchmarks, generated by AE-VAL, range from 1 to 8 (median is 3, geometric mean is 2.3).

The specifications of the system were enhanced by the designer with sets of examples that describe some additional features of the desired implementations (thus, the Skolem terms generated for the original specification might no longer be valid for the corresponding enhanced specifications). We considered 32 unique examples to enhance each benchmark. The depths of straightforward solutions, generated by AE-VAL for these benchmarks, range from 1 to 106 (median is 37, geometric mean is 32). In contrast, the depths of the solutions by our AE-VAL-PBE for these benchmarks are an order of magnitude smaller, i.e., they range from 1 to 17 (median is 5, geometric mean is 5.5). Thus, the AE-VAL-PBE was shown to be *more effective when computing compact solutions*: the ratio between depths ranges from 1 to 24, median is 6.8, geometric mean is 5.8. The synthesis time for producing the default local Skolem terms, as well as the straightforward decision trees was negligible.

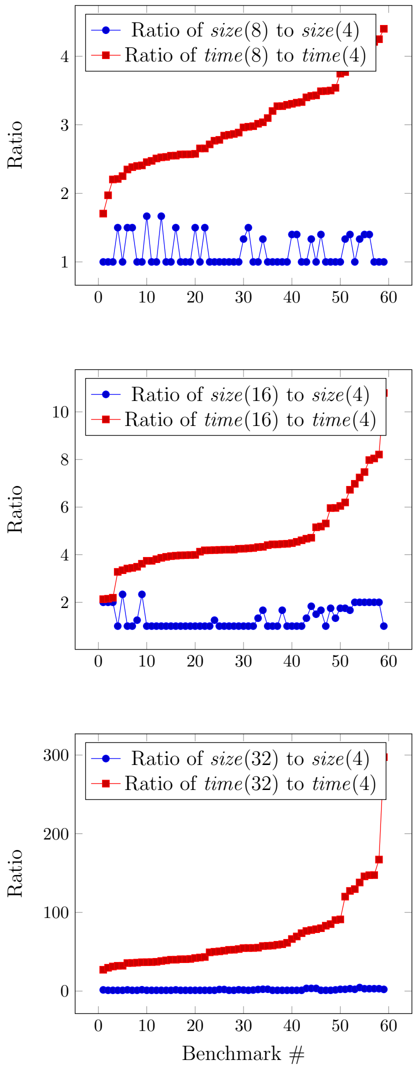

Fig. 1: Stability of our solutions.

Effect of number of examples. A common characteristic exhibited by the "plain" AE-VAL when enhancing relational specifications with examples is the growth of the resulting decision trees. Intuitively, the more examples are given, the larger solutions are generated. In this subsection we show that such a scenario is uncommon for our approach.

We performed three additional experiments, in which we kept respectively 16, 8, and 4 given examples out of the original 32 and repeated our synthesis procedure. Although the computation of a decision tree for fewer examples is less resource-demanding, the precision of solutions remained roughly the same. For 16 examples, the median depth of the decision tree is 4.9 (geometric mean is 5). For 8 examples, the median depth is 4.4 (geometric mean is 4), and for 4 examples, the median depth is 3.9 (geometric mean is 4).

We refer to this feature of our algorithm as *stability*. Fig. 1 shows more statistics on these three experiments. For every $i \in \{8, 16, 32\}$ and for each benchmark, we computed a ratio of the decision-tree depth for i examples to the depth of the decision tree for 4 examples (shown blue). Then, we compared the two for the runtime (shown red). Intuitively, the two graphs in each plot show the growths of the solution size and the synthesis time, respectively, when increasing the number of examples.

Clearly, for most of our benchmarks, the resulting solutions have the same depths, and thus do not significantly differ from each other. For a few benchmarks, however, we witnessed certain anomalies with the solving time, which we believe can be explained by the greediness of the algorithm and a large number of computed candidate relationships. In the future, we would like our procedure to invest effort in optimizing this better.

8 Related Work

Our work is broadly related to automated synthesis as well as verification techniques that utilize decision procedures.

Synthesis Techniques. Many successful instances of the general synthesis framework are based on enumerative search, where a user-provided grammar is used to constrain the space of candidate programs, along with checking correctness with respect to a specification. These include techniques that collect input-output examples lazily, by querying the specification [2, 3]. In contrast, our approach deals with input-output examples only if they are explicitly given. More importantly, our technique does not require any additional templates or grammar. In this respect, our technique is closer to functional synthesis approaches [11,21,23] that directly formulate the synthesis tasks as quantified formulas to be solved by decision procedures. However, deriving compact implementations continues to be a challenge and provides the motivation for the new ideas developed in this paper. A compaction algorithm, employed by [3], proceeds by repairing the decision trees guided by new examples. In contrast, our approach performs a *global search*: a compact decision tree is constructed at once, by taking into account all available examples. In other words, our algorithm never revisits the upper levels of already constructed decision trees and never asks for more examples.

Another class of techniques has been successfully used for synthesis of programs *only* by examples, e.g., string and other transformations in spreadsheets [14, 26,27,31]. These often require a domain-specific grammar or some type specifications to constrain the search for programs. Since a set of examples is often incomplete in practice, some generalization in dealing with examples is useful, e.g., via interaction with the user [9] or by using machine learning techniques [5, 8, 25]. We are inspired by the success of these techniques and the relative ease with which users can provide examples. However, our focus is strictly on numerical domains only, and we have experimented with applications in the area of reactive synthesis [17]. As mentioned earlier, a straightforward application of existing functional synthesis techniques on such input-output examples results in large implementations. Our motivation is to find smaller implementations. We expect that our completely automated technique could be potentially used as a submodule within a broader synthesis framework targeting a richer domain.

Table constraints, which express the combinations of values of variables that are allowed or forbidden, are widely used in Constraint Programming. Several heuristics to compress tables have been proposed [6, 16, 18, 30, 32]. While the table compression task can be seen as a generalization of our PBE task, none of these approaches proceeds further and generates an implementation from the compressed tables.

Verification Techniques. Our technique for finding symbolic linear relationships among examples is similar to techniques [10, 13, 20, 22, 24] for synthesis of invariants in program verification. In particular, these techniques can generate formulas from concrete values of program variables while discovering inductive invariants of loops. In this line of work, various feasible paths are obtained using execution or symbolic execution to generate data with values of all variables. An invariant requires generating a relation over all program variables that transit through the loop. Functional synthesis tasks, such as the one we are solving, aim at embedding a function into this relation, thus requiring more work.

9 Conclusions

We have presented a novel approach to synthesis that leverages PBE specifications and uses an FS framework for LIA. Our approach discovers preconditions and local Skolem terms by iterative partitioning of the set of examples into subsets. Each subset is described using detected relationships over inputs and outputs, which are directly used in the resulting implementations. The approach is easily extendable to deal with hybrid tasks, which are formulated in part by examples and in part by FS specifications. Our implementation on top of AE-VAL exhibits a promising performance on a set of reactive synthesis benchmarks enhanced with examples. Decision trees produced by our tool are an order of magnitude smaller than ones produced by the "plain" AE-VAL. In the future, we would like to extend this approach to other theories, such as arrays, strings, and algebraic data types, as well as to adopt more advanced ordering criteria and strategies for solution counting [4].

Acknowledgments. This work was supported in part by NSF Grant 1525936. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF.

References

1. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
2. R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In *CAV Part II*, volume 9207 of *LNCS*, pages 163–179. Springer, 2015.
3. R. Alur, A. Radhakrishna, and A. Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS, Part I*, volume 10205 of *LNCS*, pages 319–336, 2017.
4. N. Beldiceanu and H. Simonis. A Constraint Seeker: Finding and Ranking Global Constraints from Examples. In *CP*, volume 6876 of *LNCS*, pages 12–26. Springer, 2011.
5. S. Bhatia, P. Kohli, and R. Singh. Neuro-symbolic program corrector for introductory programming assignments. In *ICSE*, pages 60–70. ACM, 2018.
6. B. L. Charlier, M. T. Khong, C. Lecoutre, and Y. Deville. Automatic Synthesis of Smart Table Constraints by Abstraction of Table Constraints. In *IJCAI*, pages 681–687. ijcai.org, 2017.
7. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
8. J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. RobustFill: Neural program learning under noisy I/O. In *ICML*, volume 70, pages 990–998. PMLR, 2017.
9. D. Drachsler-Cohen, S. Shoham, and E. Yahav. Synthesis with abstract examples. In *CAV, Part I*, volume 10426 of *LNCS*, pages 254–278. Springer, 2017.
10. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458. ACM, 2000.
11. G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but Effective Functional Synthesis. In *VMCAI*, volume 11388 of *LNCS*, pages 92–113. Springer, 2019.
12. G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated discovery of simulation between programs. In *LPAR*, volume 9450 of *LNCS*, pages 606–621. Springer, 2015.
13. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*, pages 170–178. ACM, 2018.
14. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.
15. K. Huang, X. Qiu, and Y. Wang. DryadSynth: A concolic SyGuS solver, 2019. `https://github.rcac.purdue.edu/cap/DryadSynth`.
16. C. Jefferson and P. Nightingale. Extending Simple Tabular Reduction with Short Supports. In *IJCAI*, pages 573–579. IJCAI/AAAI, 2013.
17. A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen. Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts. In *TACAS, Part II*, volume 10806 of *LNCS*, pages 176–193. Springer, 2018.
18. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *CP*, volume 4741 of *LNCS*, pages 379–393. Springer, 2007.
19. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014.
20. S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

21. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.
22. S. Padhi, R. Sharma, and T. D. Millstein. Data-driven precondition inference with learned features. In *PLDI*, pages 42–56. ACM, 2016.
23. A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV*, volume 9206 of *LNCS*, pages 198–216. Springer, 2015.
24. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.
25. R. Singh. BlinkFill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.
26. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, volume 7358 of *LNCS*, pages 634–651. Springer, 2012.
27. R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *CAV, Part I*, volume 9206 of *LNCS*, pages 398–414. Springer, 2015.
28. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.
29. E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541. ACM, 2014.
30. H. Verhaeghe, C. Lecoutre, Y. Deville, and P. Schaus. Extending Compact-Table to Basic Smart Tables. In *CP*, volume 10416 of *LNCS*, pages 297–307. Springer, 2017.
31. X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.
32. W. Xia and R. H. C. Yap. Optimizing STR algorithms with tuple compression. In *CP*, volume 8124 of *LNCS*, pages 724–732. Springer, 2013.