Lazy but Effective Functional Synthesis

Grigory Fedyukovich[1(✉)], Arie Gurfinkel[2], and Aarti Gupta[1]

[1] Princeton University, Princeton, USA
{grigoryf,aartig}@cs.princeton.edu
[2] University of Waterloo, Waterloo, Canada
agurfinkel@uwaterloo.ca

Abstract. We present a new technique for generating a function implementation from a declarative specification formulated as a ∀∃-formula in first-order logic. We follow a classic approach of eliminating existential quantifiers and extracting Skolem functions for the theory of linear arithmetic. Our method eliminates quantifiers *lazily* and produces a synthesis solution in the form of a decision tree. Compared to prior approaches, our decision trees have fewer nodes due to deriving theory terms that can be shared both within a single output as well as across multiple outputs. Our approach is implemented in a tool called AE-VAL, and its evaluation on a set of reactive synthesis benchmarks shows promise.

1 Introduction

The task of generating a function implementation from a specification of an input-output relation is commonly addressed by *functional synthesis*. Many prior approaches have been proposed for functional synthesis [10,13,16,18,20], with applications in various stages of software development, from prototyping to maintaining and repairing existing products. However, there is still a great need to make the synthesizers more robust and scalable, and the synthesized implementations more compact. We build this work on recent advances in *lazy* quantifier elimination methods [3,6,14,17,20] that enabled us to progress in both these dimensions.

Synthesis tasks are often formulated as quantified formulas. We consider formulas of the form $\forall \vec{x} \,.\, \exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$ (or ∀∃-formulas in short[1]). A simple example of a synthesis task formulated as a ∀∃-formula to generate a `max`-function is shown below, where the two input variables x_1 and x_2 are universally quantified and the output y is existentially quantified:

$$\forall x_1, x_2 \,.\, \exists y \,.\, y \geq x_1 \wedge y \geq x_2 \wedge (x_1 = y \vee x_2 = y)$$

The validity of this formula means that there always exists a maximum between two integers. A witness to the maximum value, i.e., a Skolem function $y = ite(x_1 \geq x_2, x_1, x_2)$, can then be generated (and suitably decoded as a statement in a program). In this paper, we consider the general case of synthesis of *multi-output* programs, i.e., with an arbitrary number of outputs. An

[1] Here and later, we use the vector notation to denote multiple variables.

C. Enea and R. Piskac (Eds.): VMCAI 2019, LNCS 11388, pp. 92–113, 2019.
https://doi.org/10.1007/978-3-030-11245-5_5

example task is to generate a program that invokes both `max` and `min`-functions at the same time. An encoding of this task as a ∀∃-formula is somewhat bulky, as shown below:

$$\forall x_1, x_2 \,.\, \exists y_1, y_2 \,.\, y_1 \geq x_1 \wedge y_1 \geq x_2 \wedge (x_1 = y_1 \vee x_2 = y_1) \wedge \\ y_2 \leq x_1 \wedge y_2 \leq x_2 \wedge (x_1 = y_2 \vee x_2 = y_2)$$

However, a solution for this synthesis task can still be formulated concisely: $ite(x_1 \geq x_2, y_1 = x_1 \wedge y_2 = x_2, y_1 = x_2 \wedge y_2 = x_1)$. In particular, note that the predicate $x_1 \geq x_2$ is *shared* between the two outputs y_1 and y_2 in the program, which respectively denote the values of the `max` and `min` functions.

Our synthesis procedure generates an implementation of a function *while eliminating existential quantifiers* in the formula, similar to prior work by Kuncak et al. [16]. However, quantifier elimination is an expensive iterative procedure in general. To lower the overall cost of functional synthesis, we propose to use a lazy procedure [6] for quantifier elimination in ∀∃-formulas using Model-Based Projection (MBP) for linear arithmetic [14]. Unlike the prior work, our procedure does not require converting the formula into Disjunctive Normal Form (DNF), and thus often produces smaller and non-redundant implementations.

Along with the use of MBPs, we formulate criteria for an *effective decomposition* of a functional synthesis task. In particular, we aim at searching for a *structured* synthesis solution in the form of a decision tree, where each of the synthesis subtasks is defined in terms of a *precondition* and a set of *Skolem constraints* in a grammar, from which a function implementation is generated. While our notion of a precondition is similar to that in prior work [16], our MBP-based procedure results in fewer number of synthesis subtasks, thereby providing performance improvements and smaller implementations.

Our effective decomposition further enables optimization procedures for on-the-fly compaction of the generated function. In particular, we derive Skolem terms that can be re-used across multiple preconditions for a single output, and share the preconditions in a common decision tree across multiple outputs in the program. Our method identifies theory terms that can be shared both within and across outputs. While the motivation for such sharing is similar to optimization of Boolean gate-level circuits in the area of *logic synthesis*, our compaction is enabled by theory-specific reasoning (validity checks), not Boolean optimization at the propositional level. Our evaluation in a tool called AE-VAL demonstrates the benefits of our compaction algorithm, which further reduces the size of the resulting implementations by an average of two.

We have implemented our ideas in AE-VAL on top of our prior work [6], which described a procedure for determining the validity of ∀∃-formulas using MBPs for linear arithmetic [14]. The focus of that effort was on deriving Skolem witnesses for a simulation relation between two given programs. However, there was no method described for functional synthesis, which requires deriving a Skolem *function* rather than a Skolem *relation*. Furthermore, it did not consider minimization or compaction of the generated implementations. Note again, that this minimization/compaction is not at the propositional level, but requires theory-specific reasoning. The required validity checks for compaction are built

into the synthesis procedure and use the same MBP-based validity checker recursively. We provide a detailed evaluation of our tool on a selection of public benchmarks from SyGuS-COMP[2] and benchmark examples for reactive synthesis from Assume-Guarantee contracts [13].

We start by providing some background in Sect. 2. Next, in Sect. 3, we describe our criteria for effective decomposition and the MBP-based procedure for formulating the synthesis subtasks. In Sect. 4, we present a method for extracting Skolem functions from Skolem constraints. In Sect. 5, we describe our algorithm for compaction and re-use of theory terms within and across subtasks. We have implemented our procedure for functional synthesis for linear arithmetic and present a detailed evaluation in Sect. 6. Related work is described in Sect. 7 and conclusions in Sect. 8.

2 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:

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

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:

$$formula ::= true \mid false \mid p(term, \ldots, term) \mid Bvar \mid$$
$$\neg formula \mid formula \wedge formula \mid formula \vee formula$$

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 theories of Linear Rational Arithmetic (LRA) and Linear Integer Arithmetic (LIA). In LRA, $\mathcal{S} \stackrel{\text{def}}{=} \{\mathbb{Q}, Bool\}$, $\mathcal{F} \stackrel{\text{def}}{=} \{+, \cdot\}$, where $\cdot$ is a scalar multiplication (i.e., it does not allow multiplying two terms which both contain variables), and $\mathcal{P} \stackrel{\text{def}}{=} \{=, >, <, \geq, \leq, \neq\}$. In LIA, $\mathcal{C} \stackrel{\text{def}}{=} \{\mathbb{Z}, Bool\}$, $\mathcal{F} \stackrel{\text{def}}{=} \{+, \cdot, div\}$, where *div* is an integer division[3] , and $\mathcal{P} \stackrel{\text{def}}{=} \{=, >, <, \geq, \leq, \neq\}$. For both LRA and LIA, we use a shortcut $ite(x, y, z) \stackrel{\text{def}}{=} (x \wedge y) \vee (\neg x \wedge z)$, but do not include *ite* in $\mathcal{F}$.

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* (denoted $m \models \varphi$); 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 $true \implies \varphi$.

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

[2] http://sygus.seas.upenn.edu/SyGuS-COMP2018.html.

[3] We do not consider the modulo operation in this work, but our approach can be extended to support it.

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 $sk_y(\vec{x})$ is called a *Skolem term*, if $\psi(\vec{x}, sk_y(\vec{x}))$ is valid. More generally, for a valid formula $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$ over a vector of existentially quantified variables $\vec{y}$, there exists a vector of individual Skolem terms for every variable $\vec{y}[j]$, where $0 < j \leq N$ and $N = |\vec{y}|$, such that:

$$true \implies \psi(\vec{x}, sk_{\vec{y}[1]}(\vec{x}), \ldots, sk_{\vec{y}[N]}(\vec{x})) \tag{1}$$

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.

3 Decomposing Functional Synthesis

A functional synthesis task aims at generating a function from a given input-output relation. We view this in terms of validity checking of $\forall\exists$-formulas and derive Skolem terms for the existentially-quantified variables. We propose to discover Skolem terms in stages: an original task is decomposed into subtasks, where each of the subtasks is solved in isolation, and the solution to the original problem is obtained as one common *decision tree* that combines the results from the subtasks.

3.1 Illustrative Example

Consider a given formula in Disjunctive Normal Form (DNF) (we defer a discussion of a general case until later in this section). Here, it is intuitively easy to see that the individual Skolem function for each $\vec{y}[j]$ can be represented in the form of a decision tree, as illustrated in the following example.

Example 1. Given a formula $\exists y_1, y_2 \,.\, \psi(x, y_1, y_2)$ in LIA, where

$$\psi(x, y_1, y_2) \stackrel{\text{def}}{=} (x \leq 2 \wedge y_1 > -3 \cdot x \wedge y_2 < x) \vee (x \geq -1 \wedge y_1 < 5 \cdot x \wedge y_2 > x)$$

The formula is valid, which means that for every value of x there exist values of y_1 and y_2 that make either of two disjuncts true. Intuitively, the disjuncts correspond to two cases, when $x \leq 2$ or $x \geq -1$. We call these formulas *preconditions*.

To extract Skolem terms for y_1 and y_2, this example permits considering two preconditions in isolation (however, it may not be true for other formulas, see Sect. 3.3). That is, if $x \leq 2$, then y_1 should satisfy $y_1 > -3 \cdot x$ and y_2 should satisfy $y_2 < x$. In other words, the following two formulas are valid:

$$(x \leq 2) \implies \exists y_1 \,.\, (y_1 > -3 \cdot x)$$
$$(x \leq 2) \implies \exists y_2 \,.\, (y_2 < x)$$

Skolem terms for y_1 and y_2 assuming $x \leq 2$ could be $-3 \cdot x + 1$ and $x - 1$ respectively. Similarly, for the second precondition:

$$(x \geq -1) \implies \exists y_1 \,.\, (y_1 < 5 \cdot x)$$
$$(x \geq -1) \implies \exists y_2 \,.\, (y_2 > x)$$

Assuming $x \geq -1$, a Skolem term for y_1 could again be $-3 \cdot x + 1$, but a Skolem term for y_2 is $x + 1$. Combining these Skolem terms for both preconditions, we get Skolem terms for $\exists y_1, y_2 \,.\, \psi(x, y_1, y_2)$:

$$sk_{y_1}(x) \stackrel{\text{def}}{=} -3 \cdot x + 1$$

$$sk_{y_2}(x) \stackrel{\text{def}}{=} ite(x \leq 2, x - 1, x + 1)$$

Note that this composition is possible because $(x \leq 2) \vee (x \geq -1)$ is valid. In the next subsection, we describe this process formally.

3.2 Effective Decomposition

Our functional synthesis technique is based on a notion we call *effective decomposition*, defined below.

Definition 1. *A* decomposition *of a valid formula* $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$ *is a tuple* $\langle pre, \phi \rangle$, *where pre (called* preconditions*) is a vector of formulas of length* M *and* ϕ *(called* Skolem constraints*) is a matrix of dimensions* $M \times |\vec{y}|$, *such that the following three conditions hold.*

$$true \implies \bigvee_{i=1}^{M} pre[i](\vec{x}) \qquad (i\text{-totality})$$

$$pre[i](\vec{x}) \wedge \bigwedge_{j=1}^{|\vec{y}|} \phi[i,j](\vec{x}, \vec{y}) \implies \psi(\vec{x}, \vec{y}) \qquad \text{(under-approximation)}$$

$$pre[i](\vec{x}) \implies \exists \vec{y} \,.\, \bigwedge_{j=1}^{|\vec{y}|} \phi[i,j](\vec{x}, \vec{y}) \qquad (j\text{-totality})$$

Lemma 1. *For every valid formula* $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$, *a decomposition exists.*

Indeed, a decomposition could be constructed by the formula itself and a precondition *true*. We are not interested in such cases because they do not simplify a process of extracting Skolem terms from Skolem constraints ϕ. Instead, we impose additional syntactic restrictions on ϕ. In particular, we call a decomposition $\langle pre, \phi \rangle$ of $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$ $\mathcal{G}$*-effective* if all formulas ϕ are expressible in some grammar $\mathcal{G}$.

The task of extracting Skolem terms boils down to developing an algorithm that (1) produces Skolem constraints in $\mathcal{G}$, and (2) exploits $\mathcal{G}$ to extract a matrix of Skolem terms from a matrix of Skolem constraints, i.e., the following holds:

$$\vec{y}[j] = sk[i,j](\vec{x}) \implies \phi[i,j](\vec{x}, \vec{y}) \qquad \text{(embedding)}$$

Theorem 1. *Let* $\langle pre, \phi \rangle$ *be a decomposition of* $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$, *and sk be a matrix of Skolem terms, such that* (embedding) *holds. Then* Sk_j *is the Skolem term for* $\vec{y}[j]$*:*

$$Sk_j \stackrel{\text{def}}{=} ite(pre[1], sk[1,j], \ldots ite(pre[M-1], sk[M-1,j], sk[M,i])) \qquad (2)$$

A straightforward implementation of Sk_j in the form of a decision tree is shown in Fig. 1.

In this work, we restrict $\mathcal{G}$ to be the grammars of LIA/LRA (see Sect. 2) but allow neither disjunctions nor negations. In the next subsection, we outline an algorithm that creates a $\mathcal{G}$-effective decomposition while solving formulas for validity. Then, in Sect. 4, we present an algorithm for extracting Skolem terms from formulas in $\mathcal{G}$.

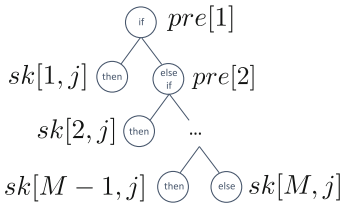


Fig. 1. A decision tree.

3.3 Witnessing Existential Quantifiers with AE-VAL

Obtaining preconditions in general requires quantifier elimination. However, it leads to expensive reasoning, which we would like to improve upon.

Example 2. Consider the following formula:

$$\exists y_1, y_2 \,.\, \big(y_1 > x_1 \vee y_2 < -x_2\big) \wedge \big(y_1 < x_2 \vee y_2 > -x_1\big)$$

If we were running the algorithm from [16], we would need to convert this formula into DNF, which would give us four disjuncts. A complete quantifier-elimination procedure would be then required to produce four preconditions and four Skolem constraints.

Our lazy quantifier-elimination method, called AE-VAL, generates both preconditions and Skolem constraints while solving the given formula for validity. In contrast to the DNF translation, for the formula in Example 2, it generates only two preconditions and two Skolem constraints.

The pseudocode of AE-VAL is shown in Algorithm 1 (we refer the reader to [6] for more detail). AE-VAL produces a sequence of Model-Based Projections (MBPs, see the definition below) [14], each of which under-approximates quantifier elimination. It iterates until the disjunction of MBPs is valid and thus avoids a complete quantifier elimination.

Definition 2. *An $MBP_{\vec{y}}$ is a function from models of $\psi(\vec{x}, \vec{y})$ to $\vec{y}$-free formulas if it has a finite image and the following hold:*

$$\text{if } m \models \psi(\vec{x}, \vec{y}) \text{ then } m \models MBP_{\vec{y}}(m, \psi)$$
$$MBP_{\vec{y}}(m, \psi) \implies \exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$$

There are different algorithms for constructing MBPs for different theories. We follow a method from [3] for LIA and present it on the following example. Intuitively, it is based on finding models, testing them on literals of the original formula, and eliminating quantifiers from the conjunctions of literals that passed the test.

Algorithm 1: AE-VAL$\Big(\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})\Big)$, cf. [6].

Input: $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$
Output: Return value $\in \{\text{VALID}, \text{INVALID}\}$ of $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$, MBPs pre, Skolem constraints ϕ

1 $M \leftarrow 1$;
2 **while true do**
3 | **if** $true \implies \bigvee_{i=1}^{M} pre[i](\vec{x})$ **then return** $\langle \text{VALID}, pre, \phi \rangle$;
4 | $tmp \leftarrow \psi(\vec{x}, \vec{y}) \wedge \bigwedge_{i=1}^{M} \neg pre[i](\vec{x})$;
5 | **if** $tmp \implies false$ **then return** $\langle \text{INVALID}, \varnothing, \varnothing \rangle$;
6 | $m \leftarrow \text{GETMODEL}(tmp)$;
7 | $\langle pre[M], \phi[M, 1], \ldots, \phi[M, |\vec{y}|] \rangle \leftarrow \text{GETMBP}(\vec{y}, m, \psi)$;
8 | $M \leftarrow M + 1$;

Example 3. Recall the formula $\exists y_1, y_2 \,.\, \psi(x_1, x_2, y_1, y_2)$ from Example 2. Its set of literals is $Lit = \{y_1 > x_1, y_2 < -x_2, y_1 < x_2, y_2 > -x_1\}$. In the first iteration, AE-VAL generates a model m_1 of ψ: $m_1 = \{x_1 \mapsto 0, x_2 \mapsto -2, y_1 \mapsto 0, y_2 \mapsto 1\}$. An MBP of ψ w.r.t. m_1 is then generated iteratively: by eliminating y_1 first, and eliminating y_2 then.

For y_1 and m_1, AE-VAL fills $\phi[1, 1]$ with a set of literals $\{l \in Lit \mid y_1 \in vars(l) \wedge m_1 \models l\}$, i.e., $\{y_1 < x_2\}$. Then AE-VAL eliminates quantifiers from $\exists y_1 \,.\, \phi[1, 1]$ and adds the result (i.e., $true$) to the MBP.

For y_2 and m_1, AE-VAL fills $\phi[1, 2]$ with $\{l \in Lit \mid y_2 \in vars(l) \wedge m_1 \models l\}$, i.e., $\{y_2 < -x_2, y_2 > -x_1\}$. It then eliminates quantifiers from $\exists y_2 \,.\, \phi[1, 2]$ and adds the result (i.e., $x_1 - x_2 > 1$) to the MBP.

Thus, after the first iteration of AE-VAL, we get the precondition $pre[1] = x_1 - x_2 > 1$, and Skolem constraints $\phi[1, 1]$ and $\phi[1, 2]$. The second iteration proceeds similarly, and AE-VAL outputs $pre[2] = true$, $\phi[2, 1] = y_1 > x_1$, and $\phi[2, 2] = y_2 > -x_1$, and terminates.

Lemma 2. *If* AE-VAL *returns* $\langle \text{VALID}, pre, \phi \rangle$ *for a formula* $\exists \vec{y} \,.\, \psi(\vec{x}, \vec{y})$*, then the formula is* effectively decomposable *by* pre *and* ϕ*, i.e.,* (i-totality)*,* (under-approximation)*, and* (j-totality) *hold.*

Intuitively, the sequence of MBPs provides a *lazy* disjunctive decomposition of the overall problem, where each precondition can capture an arbitrary subspace on the $\vec{x}$ variables (under which it is possible to derive a Skolem term for the $\vec{y}$ variables). It often requires far fewer decompositions than a DNF-based quantifier elimination approach, where each precondition can at best be a *cube*, i.e., a conjunction of predicates on $\vec{x}$. Note that the number of decompositions, M, corresponds directly to the depth of the decision tree in the generated implementations. Thus, our MBP-based procedure for quantifier elimination can

potentially perform better and lead to smaller implementations. Our experimental results in Sect. 6 show promising support.

4 Extraction of Skolem Terms

In this section, we describe our procedure for extracting individual Skolem terms from a matrix of Skolem constraints $\phi[i,j]$ in linear arithmetic. As pointed out in (embedding), this procedure is performed independently of a precondition $pre[i]$. We first describe the procedure where each $\phi[i,j]$ has occurrence of only one variable $y = \vec{y}[j]$, and thus has form $\pi(\vec{x}, y)$; and $pre[i](\vec{x}) \implies \exists y\,.\,\pi(\vec{x}, y)$ is valid. In Sect. 4.3, we describe how to handle occurrences of multiple $\vec{y}$ variables.

Although the general extraction schema is similar for all background theories, specific algorithmic details of each theory need to be discussed. In the rest of this section, we separately consider algorithms for LRA and LIA.

4.1 Skolem Terms in LRA

In Algorithm 2, we show how to extract a Skolem term for a variable $y \in \vec{y}$ from constraints having form $\pi(\vec{x}, y)$. Intuitively, Algorithm 2 constructs a graph of a function that is embedded in a relation specified by a conjunction of equalities, inequalities, and disequalities over y and $\vec{x}$. Thus, Algorithm 2 takes as input six vectors of clauses extracted from π: E, D, G, GE, L, and LE:

$$E \stackrel{\text{def}}{=} \{y = f_i(\vec{x})\}_i \quad G \stackrel{\text{def}}{=} \{y > f_i(\vec{x})\}_i \quad L \stackrel{\text{def}}{=} \{y < f_i(\vec{x})\}_i$$
$$D \stackrel{\text{def}}{=} \{y \neq f_i(\vec{x})\}_i \quad GE \stackrel{\text{def}}{=} \{y \geq f_i(\vec{x})\}_i \quad LE \stackrel{\text{def}}{=} \{y \leq f_i(\vec{x})\}_i$$

We do not consider constraints having the shape $\alpha \cdot y \sim f(\vec{x})$, because it is safe to normalize it to $y \sim \frac{f(\vec{x})}{\alpha}$ (assuming positive α; a negative α requires swapping the operator $\sim$ between $<$ and $>$, and $\leq$ and $\geq$). Finally, we assume that at least one of the vectors of clauses is non-empty, otherwise a Skolem term could be arbitrary, and there is no need to run Algorithm 2.

Below we present several helper-operators needed to construct a term sk based on a lightweight analysis of clauses in E, D, G, GE, L, and LE (where $\sim \in \{<, \leq, =, \neq, \geq, >\}$):

$$\text{ASSM}(y \sim e(\vec{x})) \stackrel{\text{def}}{=} e \qquad \text{ADD}(\ell, c) \stackrel{\text{def}}{=} \ell + c \qquad \text{MID}(\ell, u) \stackrel{\text{def}}{=} \frac{\ell + u}{2}$$

In the case when there is at least one conjunct $(y = e(\vec{x})) \in E$ (line 1), the algorithm simply returns the exact term $e(\vec{x})$. Note that there could be two or more equalities in E, which are consistent with each other due to (j-totality). Thus, it does not matter which of them is used for extracting a Skolem term.

In the case when there are lower and upper bounds (lines 2 and 3 respectively), the algorithm extracts expressions that encode the maximal and minimal values that y can take. Technically, it is done by *mapping* sets G and GE (for MAX) and L and LE (for MIN) to results of applications of ASSM to elements of these sets. In the case when $D = \varnothing$ or when ℓ and u are semantically equal, the algorithm has sufficient information to extract a Skolem term. In particular,

Algorithm 2: ExtractSkLRA$(\vec{x}, y, E, D, G, GE, L, LE)$

Input: Variable y, Skolem constraint
$\pi(\vec{x}, y) = \bigwedge_{\ell \in E \cup D \cup G \cup GE \cup L \cup LE} \ell(\vec{x}, y)$

Output: Term sk, such that $(y = sk(\vec{x})) \implies \pi(\vec{x}, y)$

1 **if** $E \neq \varnothing$ **then return** ASSM(e), s.t. $e \in E$;
2 **if** $G \cup GE \neq \varnothing$ **then** $\ell \leftarrow max(map(\text{ASSM}, G \cup GE))$;
3 **if** $L \cup LE \neq \varnothing$ **then** $u \leftarrow min(map(\text{ASSM}, L \cup LE))$;
4 **if** $\ell(\vec{x}) = u(\vec{x})$ **then return** ℓ;
5 **if** $D = \varnothing$ **then**
6 | **if** $\ell \neq \text{undef} \wedge u \neq \text{undef})$ **then return** MID(ℓ, u);
7 | **if** $\ell = \text{undef}$ **then return** ADD$(u, -1)$;
8 | **if** $u = \text{undef}$ **then return** ADD$(\ell, 1)$;
9 **else**
10 | **if** $\ell = \text{undef} \wedge u = \text{undef}$ **then** $\ell \leftarrow 1$;
11 | **if** $\ell = \text{undef}$ **then** $\ell \leftarrow$ ADD$(u, -1)$;
12 | **if** $u = \text{undef}$ **then** $u \leftarrow$ ADD$(\ell, 1)$;
13 | **return** BNSR$(\ell, u, map(\text{ASSM}, D), |D|)$;

if both lower and upper bounds are extracted, the algorithm returns a symbolic midpoint (line 6). Otherwise, it returns a symbolic value which differs from the upper or lower bounds (whichever is present) by one (lines 7 and 8).

Example 4. Consider $\pi = (y > 4 \cdot x_1) \wedge (y \geq -3 \cdot x_2 + 1) \wedge (y < x_1 + x_2)$. Algorithm 2 aims at extracting a term sk such that $\big(y = sk(x_1, x_2)\big) \implies \pi$. First, the algorithm extracts the lower bound from two inequalities with "$>$" and"$\geq$": $\ell = max(4 \cdot x_1, -3 \cdot x_2 + 1) = ite(4 \cdot x_1 > -3 \cdot x_2 + 1, 4 \cdot x_1, -3 \cdot x_2 + 1)$. Second, the algorithm extracts the upper bound from the only "$<$"-inequality: $u = x_1 + x_2$. Finally, the algorithm extracts and returns the symbolic midpoint between ℓ and u. That is, $sk = \frac{ite(4 \cdot x_1 > -3 \cdot x_2 + 1, 4 \cdot x_1, -3 \cdot x_2 + 1) + (x_1 + x_2)}{2}$.

The rest of the algorithm handles disequalities, i.e., in the case when $D \neq \varnothing$ (line 9). It assumes that ℓ and u are extracted, otherwise any suitable ℓ and u could be selected (in lines 10–12, we use some particular but not the only possible choice).

Intuitively, if y is required to differ from some $h(\vec{x})$ and to be in a range (ℓ, u), it is sufficient to pick two distinct terms v_1 and v_2 such that:

$$(y = v_1(\vec{x})) \implies (\ell(\vec{x}) < y < u(\vec{x}))$$
$$(y = v_2(\vec{x})) \implies (\ell(\vec{x}) < y < u(\vec{x}))$$
$$(v_1(\vec{x}) = v_2(\vec{x})) \implies false$$

Since each variable assignment m to $\vec{x}$ makes at most one formula from set $\{h(m) = v_1(m), h(m) = v_2(m)\}$ true, we can always extract a Skolem term $sk = ite(h = v_1, v_2, v_1)$ that satisfies $(y = sk(\vec{x})) \implies (y \neq h(\vec{x}))$.

A similar reasoning is applied to any set D of disequalities: it is enough to consider $|D| + 1$ terms, which are semantically distinct. Our algorithm can be

parametrized by any routine that extracts semantically distinct terms belonging to a range between ℓ and u. Two of possible routines are inspired respectively by a binary search (which is used in line 13) and a linear scan.

Definition 3. *Let n be the number of disequalities in D and H be the set of right sides of expressions of D, then the binary-search helper-operator is defined as follows:*

$$\mathrm{BNSR}(\ell, u, H, n) \stackrel{\text{def}}{=} \begin{cases} \mathrm{MID}(\ell, u) & \text{if } n = 0 \\ ite\Big(\bigvee\limits_{h \in H} \mathrm{MID}(\ell, u) = h, & \\ \quad \mathrm{BNSR}\big(l, \mathrm{MID}(\ell, u), H, n-1\big), \mathrm{MID}(\ell, u)\Big) & \text{else} \end{cases}$$

Example 5. Consider $\pi = (y \neq x_1 \wedge y \neq x_2)$. Since there are no inequalities in π, the lower and upper bounds are obtained from an arbitrary range, say $(0, 1)$. Otherwise, they are computed similarly to as in Example 4. Algorithm 2 uses BNSR and returns the following Skolem term:

$$sk = ite\Big(\frac{1}{2} = x_1 \vee \frac{1}{2} = x_2, ite\Big(\frac{1}{4} = x_1 \vee \frac{1}{4} = x_2, \frac{1}{8}, \frac{1}{4}\Big), \frac{1}{2}\Big)$$

Definition 4. *Let s be some number, then*

$$\mathrm{SCAN}(\ell, s, H, n) \stackrel{\text{def}}{=} \begin{cases} \ell & \text{if } n = 1 \\ ite\Big(\bigvee\limits_{h \in H} \ell = h, \mathrm{SCAN}\big(\ell + s, s, H, n-1\big), \ell\Big) & \text{else} \end{cases}$$

Example 6. Consider formula π from Example 5, for which $H = \{x_1, x_2\}$, $\ell = 0$, and $u = 1$. A Skolem term can be compiled using the call to $\mathrm{SCAN}(\ell + \frac{u-\ell}{|H|+2}, \frac{u-\ell}{|H|+2}, H, |H|+1)$:

$$sk = ite\Big(\frac{1}{4} = x_1 \vee \frac{1}{4} = x_2, ite\Big(\frac{1}{2} = x_1 \vee \frac{1}{2} = x_2, \frac{3}{4}, \frac{1}{2}\Big), \frac{1}{4}\Big)$$

4.2 Skolem Terms in LIA

In this subsection, we present an algorithm for extracting Skolem terms in LIA. Although the flow of the algorithm is similar to the flow of the algorithm for LRA, presented in Sect. 4.1, there are two differences. First, there is no need to calculate a midpoint in the case when both a lower bound ℓ and an upper bound u are given. Instead, because (j-totality) guarantees the existence of at least one integer value for all y, it is enough to choose either the least or the greatest integer value within the range (ℓ, u). Second, there are divisibility constraints, which have to be treated more carefully. Unlike the case of LRA, we consider four vectors of clauses in the Skolem constraints π over LIA:

$$E \stackrel{\text{def}}{=} \{\alpha \cdot y = f_i(\vec{x})\}_i \quad G \stackrel{\text{def}}{=} \{\alpha \cdot y > f_i(\vec{x})\}_i$$
$$D \stackrel{\text{def}}{=} \{\alpha \cdot y \neq f_i(\vec{x})\}_i \quad LE \stackrel{\text{def}}{=} \{\alpha \cdot y \leq f_i(\vec{x})\}_i$$

Algorithm 3: ExtractSkLIA$(\vec{x}, y, E, G, LE, D)$

Input: Variable y, Skolem constraint
$\pi(\vec{x}, y) = \bigwedge_{\ell \in E \cup G \cup LE \cup D} \ell(\vec{x}, y)$
Output: Term sk, such that $(y = sk(\vec{x})) \implies \pi(\vec{x}, y)$

```
1  if (E ≠ ∅) then return ASSM_ℤ(e), s.t. e ∈ E;
2  if (G ≠ ∅) then ℓ ← max(map(ASSM_ℤ, G));
3  if (LE ≠ ∅) then u ← min(map(ASSM_ℤ, LE));
4  if (D = ∅) then
5  |  if (ℓ ≠ undef) then return ADD(ℓ, 1);
6  |  if (u ≠ undef) then return u;
7  else
8  |  if (ℓ = undef ∧ u = undef) then ℓ ← 0;
9  |  if (ℓ = undef) then ℓ ← ADD(u, −1 · |D|);
10 |  return SCAN_ℤ(ℓ, D, |D|);
```

We can safely avoid clauses containing $<$ and $\geq$ because of the following transformations:

$$\frac{A < B}{A \leq B - 1} \qquad \frac{A \geq B}{A > B - 1} \tag{3}$$

We need these rules to simplify the normalization of inequalities by dividing their right sides by α (assuming positive α; a negative α requires changing the operator $\sim$ accordingly). For example, it would not be correct to normalize an inequality $5 \cdot y \geq 9$ to $y \geq div(9, 5)$. Instead, when $5 \cdot y \geq 9$ is rewritten to $5 \cdot y > 8$, the normalization works correctly: $y > div(8, 5)$. Similarly, an inequality $5 \cdot y < 9$ should be rewritten to $5 \cdot y \leq 8$ and normalized to $y \leq div(8, 5)$.

We also rewrite the divisibility constraints (i.e., $div(y, \alpha) \sim f(\vec{x})$) using the following transformations (in addition to applying (3)):

$$\frac{div(y, \alpha) = f(\vec{x})}{\alpha \cdot f(\vec{x}) \leq y < \alpha \cdot f(\vec{x}) + \alpha} \qquad \frac{div(y, \alpha) > f(\vec{x})}{y > \alpha \cdot f(\vec{x}) + \alpha - 1}$$

$$\frac{div(y, \alpha) \neq f(\vec{x})}{\bigwedge_{i=0}^{\alpha-1} y \neq \alpha \cdot f(\vec{x}) + i} \qquad \frac{div(y, \alpha) \leq f(\vec{x})}{y \leq \alpha \cdot f(\vec{x}) + \alpha - 1}$$

An example for applying the first rule is $div(y, 3) = 0$: y could be either 0, 1, or 2; or in other words $0 \leq y \wedge y < 3$. For the second rule, an example is $div(y, 3) > 0$: y could be anything greater or equal than 3, or alternatively greater than 2. Similarly, $div(y, 3) \leq 0$ is equivalent to $y \leq 2$. Finally, the rule for disequalities enumerates a finite number (equal to α) of disequalities of form $y \neq f(x)$. For instance, $div(y, 3) \neq 1$ is equivalent to $y \neq 3 \wedge y \neq 4 \wedge y \neq 5$.

The pseudocode of the algorithm that extracts a Skolem term for π in LIA is shown in Algorithm 3. It handles constraints using the following helper-operators.

$$\mathrm{ASSM}_{\mathbb{Z}}(\alpha \cdot y \sim f(\vec{x})) \stackrel{\text{def}}{=} div(f, \alpha)$$

Definition 5. *Let $h[y/\ell]$ denote the term h with term ℓ substituted for variable y. Then a helper-operator for the linear scan in LIA is implemented as follows.*

$$\mathrm{SCAN}_{\mathbb{Z}}(\ell, H, n) \stackrel{\text{def}}{=} \begin{cases} \ell & \text{if } n = 0 \\ ite\Big(\bigwedge\limits_{h \in H} h[y/\ell],\ \ell,\ \mathrm{SCAN}_{\mathbb{Z}}\big(\ell + 1, H, n - 1\big)\Big) & \text{else} \end{cases}$$

For the case when there exists an equality $\alpha \cdot y = e(\vec{x})$ in π, it is sufficient to extract $div(e, \alpha)$ for sk, because requirement (j-totality) guarantees that e is divisible by α. This is implemented in function $\mathrm{ASSM}_{\mathbb{Z}}$. To handle disequalities, the algorithm can only perform a linear scan, i.e., starting from the lower bound to make the least possible increments (i.e., by one). As opposed to the binary search, the linear scan guarantees that enough semantically distinct terms are considered. We illustrate this in the following example.

Example 7. Consider $\pi = (5 \cdot y \neq 4 \cdot x)$. Since there is no lower bound and no upper bound, we allow $\ell = 0$ (alternatively, any other term can be chosen). Then, since π has only one disequality, we get the final Skolem as a single if-then-else: $ite\Big((5 \cdot 0 \neq 4 \cdot x), 0, 1\Big)$.

4.3 Putting It All Together

Theorem 2. *For some i, let $\phi[i, j](\vec{x}, \vec{y})$ be in LRA (resp. LIA), and $|\vec{y}| = N$. Then for each $j \in [1, N]$, Algorithm 2 (resp. Algorithm 3.) extracts a term $sk[i, j]$, such that* (embedding) *holds.*

For proving this theorem, it remains to show how we obtain $\pi(\vec{x}, \vec{y}[j])$ that Algorithm 2 (resp. Algorithm 3) should take as input with each $\vec{y}[j]$. Indeed, the MBPs constructed in Sect. 3.3 allow occurrences of multiple variables from $\vec{y}$ in a clause in $\phi[i, j]$. However, by construction, a variable $\vec{y}[j]$ can appear in all $\phi[i, k], 1 \leq k \leq j$, but a variable $\vec{y}[j]$ cannot appear in any $\phi[i, k], j < k \leq N$. In particular, term $\phi[i, N]$ is only over the variables $\vec{x}$ and $\vec{y}[N]$. Therefore, we first apply Algorithm 2 (resp. Algorithm 3) to $\phi[i, N]$, to derive the Skolem term $sk[i, N]$. It is then substituted in all appearances of $\vec{y}[N]$ in other constraints $\phi[i, N - 1], \ldots, \phi[i, 1]$. Continuing such reasoning over the remaining variables leads to obtaining suitable inputs for Algorithm 2 (resp. Algorithm 3) and each $\vec{y}[j]$.

5 Synthesis of Compact Skolem Terms

Recall that Theorem 1 gives a way to construct a global Skolem term from preconditions and relations, and Sect. 4 describes algorithms to extract a local

term $sk[i,j]$ from a relation $\phi[i,j]$. So far, this provides a procedure that invokes Algorithm 2 or Algorithm 3 as soon as possible, i.e., when $\phi[i,j]$ has just been produced by the MBP-based procedure AE-VAL together with some $pre[i]$. However, for large formulas it is often the case that the number M of generated MBPs is large, and so is a vector of tuples $\langle pre[i], \bigwedge_{0<j\leq N} \phi[i,j]\rangle$ where $0 < i \leq M$.

In this section, we propose to leverage the output of AE-VAL for producing compact Skolem terms. We first describe how to reduce the number of distinct Skolem terms among the tuples generated by AE-VAL for each $y \in \vec{y}$. Next, we aim to reduce the depth of the overall decision tree in case of multiple outputs $\vec{y}$, i.e., extracting a common if-then-else (ite) block which is shared among *all* outputs.

5.1 Optimizing Decision Trees by Combining Preconditions

Our goal is to decrease the depth of a decision tree that combines Skolem terms for M different preconditions. Recall that at each node i, where $0 < i \leq M$ and for variable $y = \vec{y}[j]$, the Skolem term $sk[i,j]$ and the precondition $pre[i]$ should be connected via (j-totality) and (embedding), i.e.:

$$pre[i](\vec{x}) \implies \exists y\,.\,\phi[i,j](\vec{x},\vec{y})$$
$$y = sk[i,j](\vec{x}) \implies \phi[i,j](\vec{x},\vec{y})$$

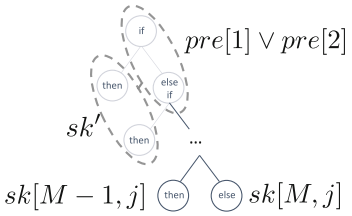


Fig. 2. Optimized decision tree.

Note that preconditions in the decision tree could potentially guard the same Skolem terms, in which case we could compact the size. This is illustrated pictorially in Fig. 2 for the example shown earlier in Fig. 1. In particular, if $sk[1,j]$ and $sk[2,j]$ in Fig. 1 could be replaced by a common sk', then the preconditions $pre[1]$ and $pre[2]$ can be merged using a disjunction, thereby decreasing the depth of the decision tree. The challenge is that sk' might not necessarily be obtained by Algorithm 2 or Algorithm 3, because Skolem constraints $\phi[1,j]$ or $\phi[2,j]$, taken in isolation, are in general not restrictive enough. However, sk' could be produced by Algorithm 2 or Algorithm 3 if $\phi[1,j] \wedge \phi[2,j]$ is given as input.

Generalizing this idea further, we consider an expensive minimization algorithm to search over all partitions of the set $\mathbb{M} \stackrel{\text{def}}{=} \{1,\ldots,M\}$ and find the *best* partition such that each index in a class of the partition can share the same Skolem term. More formally, for each partition $\mathbb{P} \stackrel{\text{def}}{=} \{p_1,\ldots p_r\}$ of $\mathbb{M}$, for each class p_k in the partition, we check that:

$$\bigvee_{i\in p_k} pre[i](\vec{x}) \implies \exists y\,.\,\bigwedge_{i\in p_k} \phi[i,j](\vec{x},y) \tag{4}$$

If all r implications hold, then $\mathbb{P}$ is a *valid candidate partition*, associated with r Skolem terms $sk',\ldots,sk^{(r)}$ derived from $\bigwedge_{i\in p_1} \phi[i,j](\vec{x},y)$, $\ldots$, $\bigwedge_{i\in p_r} \phi[i,j](\vec{x},y)$, respectively. We then select the best partition among the valid candidate par-

Algorithm 4: GETPARTITIONCLASS($\mathbb{I}, pre, \phi, y = \vec{y}[j]$)

Input: Initial set of indices $\mathbb{I}$, preconditions pre, constraints ϕ
Output: Output set of indices p_k

1 $p_k \leftarrow \mathbb{I}$;
2 **while** $\bigvee_{i \in p_k} pre[i](\vec{x}) \not\Longrightarrow \exists y \,.\, \bigwedge_{i \in p_k} \phi[i,j](\vec{x}, y)$ **do**
3 $\quad p_k \leftarrow \{i \in p_k \mid pre[i](\vec{x}) \implies \exists y \,.\, \bigwedge_{i \in p_k} \phi[i,j](\vec{x}, y)\}$;
4 **for** $i' \in \mathbb{I} \setminus p_k$ **do**
5 $\quad$ **if** $\bigvee_{i \in p_k \cup \{i'\}} pre[i](\vec{x}) \implies \exists y \,.\, \bigwedge_{i \in p_k \cup \{i'\}} \phi[i,j](\vec{x}, y)$ **then**
6 $\quad\quad p_k \leftarrow p_k \cup \{i'\}$;
7 **return** p_k;

titions, based on size of resulting Skolem terms (or other cost criteria). Clearly, examining all possible partitions would have exponential cost, with the check for each partition class also being an expensive validity check.

Instead of the expensive exact minimization, we adopt a greedy strategy for finding a *good* (but not necessarily the best) valid candidate partition, possibly within a predetermined number of iterations. The routine for identifying each partition class p_k is shown in Algorithm 4. First, p_1 is selected from $\mathbb{M}$, then p_2 is selected from $\mathbb{M} \setminus p_1$, and so on.

Algorithm 4 is based on iteratively guessing a set of indices p_k and checking an implication of the form (4). The guessing proceeds in two phases: first it checks if all eligible indices from a set $\mathbb{I} \subseteq \mathbb{M}$ are in p_k. If so, the algorithm terminates. Otherwise, it iteratively tries to strengthen the left side of (4) by removing some of the disjuncts (line 3). After removing a disjunct $pre[i']$ from the left side, the Skolem constraint $\phi[i', j]$ should also be removed from the right side, and the validity check repeats. This way, the algorithm is guaranteed to find the set of indices p_k (possibly, empty) in a finite number of iterations.

The second phase of the guessing aims at strengthening p_k. It simply traverses the set of indices $\mathbb{I} \setminus p_k$ (line 5), adds $pre[i']$ and $\phi[i', j]$ to the left and right sides of 4 respectively and checks validity. The motivation behind the second phase is that the first phase could be too aggressive in practice, thus removing more indices from p_k than needed.

Example 8. Recall our formula from Example 1. For generating a Skolem term for y_1, we create the following $\forall\exists$-formula and check its validity:

$$(x \geq -1 \vee x \leq 2) \implies \exists y_1 \,.\, (y_1 < 5 \cdot x \wedge y_1 > -3 \cdot x)$$

Since this formula is valid, our algorithm creates a single Skolem term $sk[1,1] = sk[1,2] = -3 \cdot x + 1$.

For generating a Skolem term for y_2, the corresponding $\forall\exists$-formula is invalid, and our algorithm generates two different Skolem terms $sk[2,1]$ and $sk[2,2]$:

$$(x \geq -1 \vee x \leq 2) \implies \exists y_2 \,.\, (y_2 < x \wedge y_2 > x)$$

5.2 Minimizing the Depth of the Common Decision Tree

To allow re-use of theory terms among multiple outputs $\vec{y}$, a common *ite*-block could be pulled outside of the individual decision trees for each output, denoted $Sk_{\vec{y}}(\vec{x}, \vec{y})$:

$$Sk_{\vec{y}}(\vec{x}, \vec{y}) \stackrel{\text{def}}{=} ite\Big(pre[1](\vec{x}), \bigwedge_{j=1}^{N} \vec{y}[j] = sk[1, j](\vec{x}), \ldots$$
$$ite\big(pre[M-1](\vec{x}), \bigwedge_{j=1}^{N} \vec{y}[j] = sk[M-1, j](\vec{x}), \bigwedge_{j=1}^{N} \vec{y}[j] = sk[M, i](\vec{x})\big)\Big)$$

In general, depending on the cost criteria, it may be advantageous to not have a common *ite*-block at all or to have it be common to a subset of the outputs rather than all outputs. In this section, we consider a simple case where a common *ite*-block is shared among *all* outputs. Then, the remaining goal is to reduce the depth of this block by finding redundant branches.

Recall that Algorithm 4 can be used *per output* to find a good partition among the tuples, i.e., to decide which branches of the *ite*-block can share the same Skolem term. We view the results from this algorithm in the form of a matrix of Skolem terms, with a row for each *ite*-branch and a column for each output. Then, it is straightforward to identify redundant branches, which correspond to identical rows in the matrix. We illustrate this process in an example.

Example 9. Consider a formula with four existentially quantified variables $\vec{y}$ and four preconditions. Suppose the algorithm from Sect. 5 returns the partitions of the set $\{1, 2, 3, 4\}$ for each variable in $\vec{y}$, as shown in the following matrix.

For instance, $\vec{y}[1]$ requires a partition $\{p_1\}$ where $p_1 = \{1, 2, 3, 4\}$. Variable $\vec{y}[2]$ requires partition $\{q_1, q_2\}$ where $q_1 = \{1\}$ and $q_2 = \{2, 3, 4\}$. Variable $\vec{y}[3]$ requires partition $\{r_1, r_2\}$ where $r_1 = \{1, 2, 4\}$ and $q_2 = \{3\}$. Variable $\vec{y}[4]$ requires partition $\{s_1, s_2, s_3\}$ where $s_1 = \{1\}$, $s_2 = \{2, 4\}$ and $s_3 = \{3\}$.

We can easily identify identical rows $A_1, \ldots A_k$ in the matrix, such that for all $0 < j < M$, elements $A_1[j] = A_2[j] = \ldots = A_k[j]$ are equal.

	$\vec{y}[1]$	$\vec{y}[2]$	$\vec{y}[3]$	$\vec{y}[4]$
$pre[1]$	p_1	q_1	r_1	s_1
$pre[2]$	p_1	q_2	r_1	s_2
$pre[3]$	p_1	q_2	r_2	s_3
$pre[4]$	p_1	q_2	r_1	s_2

In this example, row A_1 corresponds to $pre[2]$, and row A_2 corresponds to $pre[4]$. Thus, individual Skolem terms for all variables for $pre[2]$ and $pre[4]$ can be combined, and the depth of the common *ite*-block is reduced by one.

6 Evaluation

We implemented our synthesis algorithms on top of the AE-VAL tool [6] which is in turn built on top of the Z3 SMT solver [5]. Note that the previous imple-

mentation of AE-VAL was already able to solve quantified formulas for validity via an iterative MBP construction and to extract Skolem constraints. However, it did not provide procedures to extract Skolem functions (described in Sect. 4) or to compact them (described in Sect. 5). In particular, note that during our compaction procedure, we use AE-VAL recursively to solve subsidiary quantified formulas of the form (4).

6.1 Results on Benchmark Examples

We considered 134 ∀∃-formulas originated from various Assume-Guarantee contracts written in the Lustre programming language [13] [4]. The majority of benchmarks are derived from industrial projects, such as a Quad-Redundant Flight Control System, a Generic Patient Controlled Analgesia infusion pump, as well as a Microwave model, a Cinderella-Stepmother game, and several hand-written examples. Since the original set of benchmarks include minor variations of the same tasks, we identified 80 distinct benchmarks[5] for presentation in Table 1.

All the ∀∃-formulas had more than one existentially-quantified variable. Table 1 presents the statistics and results on the benchmarks. The formulas are over 5–100 universally-quantified variables and 2–49 existentially-quantified variables. The highest depth of the common *ite*-block in the produced Skolem[6] is 7. AE-VAL was able to terminate on all of them within a timeout of 60 s. The solving stage (including construction of MBPs and collecting Skolem constraints) took less than a second for all benchmarks. Compiling $\mathbf{Skolem}_1$ (i.e., without compaction) took insignificant time, but compacting $\mathbf{Skolem}_2$ took much longer for 11 outliers (the most crucial one is №16). This can be explained by many iterations for greedily finding a good partition, as explained in Sect. 5.

Figure 3 visualizes the effect of the Skolem compaction. Each point in the plot corresponds to a pair of runs of AE-VAL: the x-axis shows the size of the compacted Skolem (i.e., extracted with the use of both techniques from Sect. 5), and the y-axis shows the size of the naively created Skolem. The geometric mean for the ratio is 2.06, and the largest improvement is 6.95 – seen for the benchmark №38. In nearly half of the cases (35 out of 80), the depth of the *ite*-structure in the Skolem decreased at least by one. However, what proved to be the most effective for compaction is the factoring out of individual Skolem terms for particular variables, i.e., AE-VAL found a function which is good for all preconditions by greedy partitioning.

[4] Not to be confused with the evaluation of [13] which applied AE-VAL iteratively, and most of the formulas were invalid. Here, we considered only valid formulas and focused only on the Skolem extraction.

[5] These benchmarks are available at: http://www.cs.princeton.edu/~grigoryf/aeval-benchs.zip.

[6] Without taking into account the individual *ite*-s due to computing greatest and lowest bounds and handling disequalities, as described in Sect. 4.

Table 1. Concrete evaluation data.

$\forall\exists\,.\,\psi$: Synthesis task; $\mathbf{Skolem}_1$: without compaction, $\mathbf{Skolem}_2$: with compaction; **size**: total number of Boolean and arithmetic operators, $\#_\forall$: number of universally-quantified variables, $\#_\exists$: number of existentially-quantified variables, $\#_\Downarrow$: depth of the *ite*-block, **time**: synthesis time (in seconds, for $\mathbf{Skolem}_2$, including the compaction).

№	$\forall\exists\,.\,\psi$ size	$\#_\forall$	$\#_\exists$	$\mathbf{Skolem}_1$ size	$\#_\Downarrow$	time	$\mathbf{Skolem}_2$ size	$\#_\Downarrow$	time
1	371	71	25	192	5	0.46	80	4	3.37
2	337	71	49	116	2	0.31	61	2	0.38
3	337	70	49	106	2	0.3	64	2	0.27
4	302	100	30	175	4	0.29	95	4	1.28
5	296	39	14	180	5	0.29	61	3	2.03
6	296	98	30	175	4	0.29	95	4	1.34
7	267	90	49	168	3	0.42	77	3	0.71
8	247	36	25	126	4	0.33	39	3	0.49
9	222	39	8	128	4	0.34	54	3	0.79
10	210	52	7	27	2	0.07	20	2	0.12
11	201	51	30	231	5	0.39	124	5	4.87
12	201	50	30	130	3	0.23	77	3	0.45
13	197	21	14	58	3	0.09	15	1	0.08
14	195	33	9	199	3	0.24	77	2	0.35
15	178	30	8	101	3	0.21	44	2	0.34
16	174	10	7	321	7	0.95	195	6	91.3
17	166	39	23	552	2	0.3	232	2	0.23
18	155	52	15	151	2	0.17	67	2	0.16
19	151	23	20	115	2	0.22	64	2	0.21
20	149	11	8	240	5	0.73	128	4	3.01
21	147	24	9	260	4	0.35	87	3	0.75
22	147	23	9	120	2	0.18	69	2	0.31
23	140	31	7	34	2	0.07	27	2	0.15
24	139	30	8	139	3	0.22	37	1	0.24
25	137	47	20	89	2	0.19	49	2	0.23
26	134	22	8	210	6	0.44	74	4	12.6
27	134	21	8	54	2	0.18	36	2	0.25
28	117	36	15	151	2	0.15	53	1	0.09
29	105	22	8	290	6	0.44	61	3	9.38
30	105	21	8	138	3	0.25	66	3	0.42
31	102	71	30	176	4	0.27	99	4	1.18
32	102	32	5	21	2	0.11	14	2	0.11
33	95	20	7	94	4	0.29	39	3	0.76
34	95	19	7	72	3	0.25	44	3	0.41
35	84	33	23	552	2	0.3	232	2	0.22
36	82	26	5	21	2	0.09	14	2	0.11
37	82	25	5	21	2	0.09	14	2	0.1
38	78	21	15	431	4	0.35	62	1	0.18
39	78	20	15	107	2	0.14	41	1	0.09
40	75	25	4	165	4	0.38	147	4	0.81
41	71	37	7	26	2	0.07	19	2	0.14
42	71	37	7	34	2	0.08	27	2	0.11
43	66	13	7	181	2	0.12	75	2	0.17
44	62	18	5	21	2	0.1	14	2	0.1
45	57	17	4	121	3	0.32	109	3	0.62
46	57	15	5	34	2	0.09	11	1	0.08
47	51	10	5	36	2	0.09	25	2	0.11
48	44	13	3	14	2	0.08	11	2	0.1
49	44	12	3	14	2	0.08	11	2	0.1
50	40	18	15	429	4	0.34	99	3	0.53
51	39	12	9	24	2	0.07	18	2	0.1
52	38	10	4	218	4	0.43	197	4	1.32
53	38	9	4	88	3	0.31	76	3	0.51
54	38	13	6	175	5	0.11	28	1	0.14
55	38	8	5	30	2	0.09	10	1	0.07
56	38	9	5	44	2	0.08	12	1	0.07
57	38	12	6	36	2	0.06	17	1	0.06
58	38	11	5	69	2	0.06	32	1	0.05
59	38	10	5	51	2	0.06	22	1	0.05
60	34	12	7	260	3	0.14	75	2	0.24
61	33	13	4	24	2	0.05	10	1	0.05
62	28	9	4	47	2	0.09	38	2	0.17
63	28	8	3	12	2	0.08	9	2	0.09
64	26	8	3	11	2	0.08	8	2	0.11
65	26	8	4	42	2	0.07	11	1	0.06
66	25	5	3	14	2	0.08	11	2	0.12
67	24	14	11	26	2	0.08	15	2	0.11
68	22	8	4	28	2	0.05	13	1	0.05
69	22	7	4	20	2	0.05	9	1	0.05
70	21	6	5	13	2	0.05	8	2	0.11
71	21	16	9	24	2	0.06	18	2	0.1
72	20	13	4	198	4	0.4	177	4	1.47
73	20	15	6	181	5	0.11	28	1	0.12
74	20	10	5	44	2	0.09	12	1	0.07
75	20	14	5	71	2	0.05	32	1	0.06
76	16	5	2	13	2	0.06	11	2	0.12
77	15	7	4	47	2	0.1	34	2	0.23
78	14	9	4	42	2	0.06	11	1	0.06
79	12	8	5	14	2	0.05	9	2	0.1
80	12	9	4	30	2	0.05	13	1	0.05

6.2 Comparison with CVC4

We also compared AE-VAL with state-of-the-art tool CVC4 [20], version 1.7-prerelease [git master 464470c3], the winner of the general track of the fifth SyGuS-COMP. Like AE-VAL and unlike most of the synthesizers based on an enumerative search (e.g. [2]), the *refutation-based synthesizer* in CVC4 does not

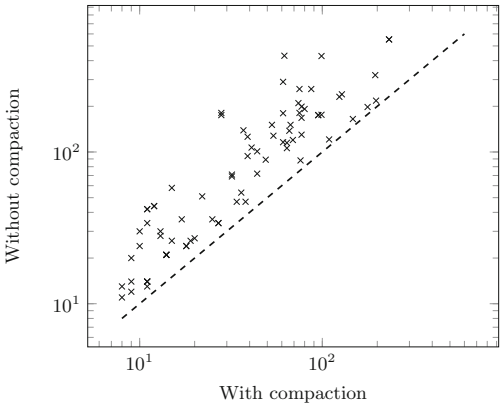
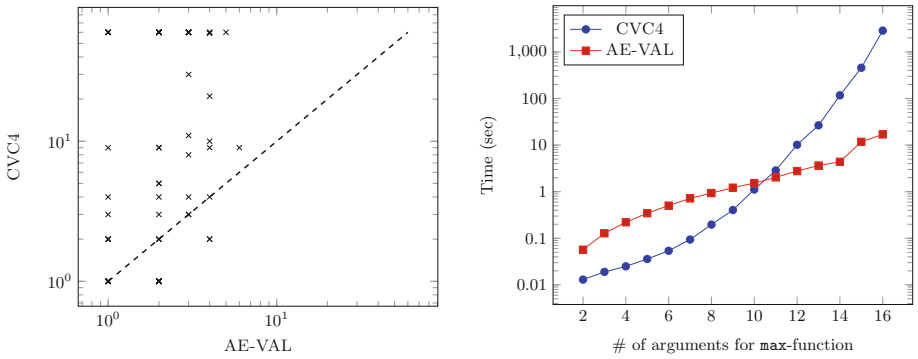


Fig. 3. Size of Skolems terms (i.e., total numbers of Boolean and arithmetic operators).



(a) *Ite*-depth for benchmarks from Table 1. **(b)** Synthesis time for `max`-functions.

Fig. 4. Benefits of AE-VAL over CVC4.

enforce any syntactic restrictions on its solutions, e.g., formal grammars or templates, and it is more efficient than an enumerative synthesizer also implemented in CVC4.

Among 80 benchmarks from Table 1, CVC4 was able to solve 55, and it exceeded a timeout of 60 s for the remaining 25 benchmarks. In Fig. 4(a), we report the ratio of depths of the *ite*-blocks generated in the implementations. In most of the cases, our implementations have shorter depths.

Note that due to reasonings of encoding [19], CVC4 solved slightly different problems, in which it extracted only individual Skolems for each output variable. It is unable to combine them in one relation or share them, as opposed to what our tool does. Thus, we are unable to compare the overall *size* of the implementations produced by CVC4 and our method.

In addition, we performed comparison experiments on isolated groups of benchmarks from SyGuS-COMP, in which the formal grammars were ignored by both tools. In particular, we considered nearly fifty *single-invocation* benchmarks from groups `array_sum`, `array_search`, and `max`. The performance of both AE-VAL and CVC4 on `array_sum` and `array_search` is similar – both tools converge in less than a second. Figure 4(b) shows a comparison of AE-VAL with CVC4 on a sequence of `max`-benchmarks, in which the number of arguments n for the function `max` being synthesized varies from 2 to 16. AE-VAL and CVC4 both converge with similar results, but the synthesis time varies significantly. Note that for $n < 10$, both tools require less than 1 s (and CVC4 is slightly faster), but for larger n, the performance of CVC4 gets worse almost exponentially, while the performance of AE-VAL remains reasonable. In particular, CVC4 is unable to synthesize a `max` function with 17 inputs after two hours, but AE-VAL synthesizes a solution in just forty seconds.

7 Related Work

Our approach follows the classical flow of functional synthesis for unbounded domains proposed in [16]. Their main idea is to enhance quantifier-elimination procedures with a routine to generate witnesses. However, in practice, it requires an expensive conversion of the specification to DNF and applying quantifier elimination for each disjunct. With our MBP-based lazy quantifier-elimination procedure AE-VAL, we have made the approach more scalable and robust, while keeping the elegance and improving generality of the witness generation procedures. Furthermore, our approach benefits from additional optimization stages to make the final implementations compact.

As mentioned earlier, an older version of AE-VAL was built and successfully used for solving the validity of ∀∃-formulas [6]. It has been successfully used in many applications:

- Realizability checking and synthesis from Assume-Guarantee contracts [13],
- Non-termination checking, and (potentially) synthesis of never-terminating lasso-shaped programs [9],
- Synthesis of simulation relations between pairs of programs [6,7],
- Synthesis of (candidates of) inductive invariants [8].

However, it did not include any procedures to generate terms for a pure functional synthesis setting, or to compact the Skolems and share terms. We believe our new procedures can further improve the above-listed and other applications of AE-VAL.

An alternative way to quantifier elimination for solving functional synthesis tasks is implemented in CVC4 [20]. Their refutation-based approach aims at determining the unsatisfiability of the negated form of ∀∃-formula. A solution is then directly obtained from an unsatisfiable set of ground instances of the negated synthesis conjecture. Similarly to AE-VAL, CVC4 proceeds lazily and creates a decision tree. However, as confirmed by our evaluation, their decision

trees are often larger than the decision trees produced by AE-VAL for the same tasks.

Laziness in the search-space exploration allows viewing both AE-VAL and CVC4 as instances of Counterexample-Guided Inductive Synthesis (CEGIS [21]). Typically, the CEGIS-based algorithms, e.g., [1,2,21,22], perform a guided search over the syntax tree of the program being synthesized. Our approach to synthesis, as well as [16] and [20], is driven by the logical structure of a background theory and does not put any restriction on the syntax tree size. This allows generating large and expressive implementations (such as `max`-functions over dozens of inputs) quickly.

There is a rich body of work on logic synthesis, i.e., synthesis of Boolean gate-level functions from specifications in propositional logic [12]. This considers synthesis of two-level (e.g., sum of products) or multi-level circuits, with minimization of various cost criteria (size, delay, power, etc.), and with sharing Boolean gates across multiple outputs. While our motivation for sharing "logic" is similar, note that we identify theory terms that can be shared within the implementation of an output, and across implementations of multiple outputs. Thus, our minimization/compaction is not at the Boolean-level, but requires theory-specific reasoning (validity checks). Furthermore, most logic synthesis efforts start with functional specifications. There have been some efforts in considering relational specifications [4,15], but these are fairly straightforward extensions of well-known functional techniques.

Finally, a procedure similar to Model-Based Projection has also been used for existential quantification in Boolean formulas [11], where it was called circuit-cofactoring. The application considered there was SAT-based model checking, where a pre-image of a given set of states is computed by existential quantification of a set of variables. The main idea was to use a model on the quantified variables to derive a circuit-cofactor (including disjunctions), which can capture many more states than a generalized cube on the remaining variables. This resulted in far fewer enumerations than cube-based enumeration techniques.

8 Conclusions

We have presented a novel approach to functional synthesis based on lazy quantifier elimination. While checking realizability of the given specification, our algorithm produces a system of synthesis subtasks through effective decomposition. Individual solutions for these subtasks generate a decision tree based implementation, which is further eligible for optimizations. Compared to the existing approaches, our generated solutions are more compact, and the average running time for their synthesis is reasonably small. We have implemented the approach in a tool called AE-VAL and evaluated it on a set of reactive synthesis benchmarks and benchmarks from SyGuS-COMP. We have identified classes of programs when AE-VAL outperformed its closest competitor CVC4 both on running time and on *ite*-depth of implementations. In the future, we wish to extend AE-VAL to other first-order theories, to support (whenever applicable)

enumeration-based reasoning, which utilizes grammars and results in even more compact solutions, and to leverage specifications enhanced with input-output examples.

Acknowledgments. We thank Andreas Katis for providing encodings of benchmarks for reactive synthesis from Assume-Guarantee contracts into an SMT-LIB2 format acceptable by AE-VAL.

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. Alur, R., Černý, P., Radhakrishna, A.: Synthesis through unification. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 163–179. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_10
2. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 319–336. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_18
3. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: LPAR (short papers), EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
4. Brayton, R.K., Somenzi, F.: An exact minimizer for boolean relations. In: ICCAD, pp. 316–319. IEEE (1989)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 606–621. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_42
7. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Property directed equivalence via abstract simulation. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 433–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_24
8. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: FMCAD. ACM (2018)
9. Fedyukovich, G., Zhang, Y., Gupta, A.: Syntax-guided termination analysis. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 124–143. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_7
10. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_22
11. Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: ICCAD, pp. 510–517. IEEE Computer Society/ACM (2004)
12. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Springer, Heidelberg (2006). https://doi.org/10.1007/b117060

13. Katis, A., et al.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 176–193. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_10
14. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
15. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 113–123. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_12
16. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT **15**(5–6), 455–474 (2013)
17. Monniaux, D.: A quantifier elimination algorithm for linear real arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 243–257. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_18
18. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989)
19. Raghothaman, M., Udupa, A.: Language to specify syntax-guided synthesis problems. CoRR, abs/1405.5590 (2014)
20. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_12
21. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
22. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: PLDI, pp. 530–541. ACM (2014)