



Synthesiz3 This: an SMT-Based Approach for Synthesis with Uncomputable Symbols

Petra Hozzová* and Nikolaj Bjørner†

*Czech Technical University, Prague, Czech Republic

†Microsoft, Redmond, USA



Abstract—Program synthesis is the task of automatically constructing a program conforming to a given specification. In this paper we focus on synthesis of single-invocation recursion-free functions conforming to a specification given as a logical formula in the presence of uncomputable symbols (i.e., symbols used in the specification but not allowed in the resulting function). We approach the problem via SMT-solving methods: we present a quantifier elimination algorithm using model-based projections for both total and partial function synthesis, working with theories of uninterpreted functions and linear arithmetic and their combination. For this purpose we also extend model-based projection to produce witnesses for these theories. Further, we present procedures tailored for the case of uniquely determined solutions. We implemented a prototype of the algorithms using the SMT-solver Z3, demonstrating their practical efficiency compared to the state of the art.

I. INTRODUCTION

Program synthesis is the task of finding a function satisfying the given specification. In this paper we focus on synthesis of recursion-free functions and require the solution to be guaranteed to be correct. To this end we lean on automated reasoning methods and do not consider machine learning. As synthesis subsumes verification, it is an inherently hard problem. Yet, recent developments in this area showed that fragments of the problem are tractable [15], [41], [39], [21].

There are many flavors of synthesis specifications. We consider specifications expressed as logical formulas capturing relational constraints on symbolically represented inputs/outputs, recently addressed by an SMT-based approach [39] and deductive synthesis in saturation [21]. Such specifications can use additional syntactic constraints: in [39] (and in the SyGuS paradigm in general [2]), the resulting function must syntactically conform to the provided grammar, while [21] introduces so-called *uncomputable* symbols which are not allowed to occur in the solution. However, while [21] uses uncomputable symbols, it does not formally characterize them. That motivated this work: we explore the semantics of specifications with uncomputable symbols, and we present an algorithm solving such specifications modulo (combinations of) theories.

Uncomputable symbols are a powerful tool allowing to make the specification more expressive, e.g. by (i) requiring the solution to be maximal, (ii) assuming the necessary condition for the existence of a total solution, or (iii) forcing the solution to discover properties of functions or predicates

used in the specification, but not allowed in the output. While [21] introduces specialized inference rules to handle the uncomputable symbols, we integrate them directly into the specification, expressed as a logical formula, by using second-order quantification. The challenges we then solve are how to *project away* these symbols from the formulas, and how to *find witnesses* capturing the solutions. To solve these challenges, we take an SMT-based route: we present an algorithm based on *quantifier elimination games* [7], [13], using and extending *model-based projections* [27], [6].

Contributions: In this work we synthesize recursion-free functions for first-order specifications using uncomputables and theories. Our main contributions are:

- 1) An algorithm for function synthesis in the presence of *uncomputable symbols* (Section III). In contrast to prior work [21], we use quantifiers to handle the scope of uncomputables. We present tailored model-based projection procedures (Section IV) used for quantifier elimination.
- 2) Our approach works directly for *partial function synthesis* (see Section III-C). That is, for specifications that are not realizable on all inputs, our approach identifies necessary and sufficient *pre-conditions* when solutions exist.
- 3) We introduce directed inferences exploiting the case when specifications have *unique solutions* (Section V).
- 4) We implemented a prototype of our method using the SMT-solver Z3 and compared it to the state of the art (Section VI).

II. PRELIMINARIES

We assume multi-sorted first-order and second-order logics and use standard definitions of their syntax and semantics. When discussing validity, we consider all formulas implicitly universally quantified. We work with theories as defined in SMT-LIB [5], in particular with the theories of equality and uninterpreted functions (EUF), and of linear integer and real arithmetic (LIA and LRA).

By $E[t]$ we signify that the term or formula E contains the term t , and by $E[t/y]$ we denote the term or formula we obtain from E by replacing every occurrence of the variable y by the term t . By \simeq we denote the equality predicate, by $:=$ assignment, by \mathbf{t} the tuple of terms (or variables/symbols) t_1, \dots, t_n , and by T_Σ the set of terms over the set of symbols Σ . We recall known notions we use in the paper: negation

normal form (NNF) of a formula F , denoted $NNF(F)$, is any formula equivalent to F , consisting only of literals, conjunctions, and disjunctions; disjunction normal form (DNF) is an NNF which is a disjunction of conjunctions of literals. We use the abbreviations sat/unsat for satisfiable/unsatisfiable. We abuse the notation and by \top, \perp we denote the logical constants for true and false, as well as their values, and in our algorithms we also denote the undefined value by \perp . We reduce predicates into functions by replacing each atom $p(t)$ by $f_p(t) \simeq \top$, where f_p is a fresh function symbol with the domain $\{\perp, \top\}$. We write $\bar{\ell}$ for the complement of the literal ℓ . When C is a conjunction of literals $\bigwedge_{\ell \in \mathcal{L}} \ell$, we interchangeably refer to it as a set of its literals \mathcal{L} . Given a set of literals \mathcal{L} and a formula F , where $\mathcal{L} \wedge F$ is unsat, we write $Core(\mathcal{L}, F)$ for $\mathcal{C} \subseteq \mathcal{L}$ such that $\mathcal{C} \wedge F$ is unsat.

We use “program” and “function” in the context of objects we synthesize as synonyms. By ϵ we denote the empty list and we use $|$ to denote list construction. We write $C \rightarrow r$ for the pair of a condition C and a term r used as one case of a program, and we represent the (possibly partial) program in the synthesis process as a list $C_1 \rightarrow r_1 \mid \dots \mid C_n \rightarrow r_n$. We construct the whole program from the list using the case constructor,

case $C_1 : r_1; \dots$ case $C_{n-1} : r_{n-1};$ default : r_n ,

which evaluates to the value r_i for $1 \leq i < n$ if $C_i \wedge \bigwedge_{j=1}^{i-1} \neg C_j$ is true, and to the value r_n otherwise. Since a list of pairs of conditions and terms uniquely defines a program, we sometimes abuse the notation and use the list in place of its corresponding program. Finally, we use $_$ to denote *any* value in the function – i.e., the case when any value of the correct sort satisfies the specification, and we use $_$ when there exists at least one computable term of that sort.

III. SYNTHESIS OF TOTAL AND PARTIAL FUNCTIONS

A. Specifications and Realizers

We focus on the class of synthesis specifications given as closed second-order logical formulas of the form

$$\exists f. \forall c, u. \varphi[f(c), c, u], \quad (1)$$

where f is the recursion-free function we want to synthesize, c are all symbols the function can depend on, and u are the symbols f should be independent of. Following the terminology of [21], we call c *computable* and u *uncomputable* symbols. The computable symbols include all symbols which are allowed in the solution for f : f ’s inputs, as well as constants, functions, and predicates. We call the terms and formulas over c *computable terms/formulas*, the terms/formulas containing symbols from u *uncomputable terms/formulas*, and the uncomputable symbols themselves also *uncomputables*.

Note that (1) specifies a *single single-invocation function*: f always occurs in φ with the same arguments c . Thus, specification (1) can be equivalently expressed in “unskolemized” (yet still possibly second-order) form as

$$\forall c. \exists y. \forall u. \varphi[y, c, u], \quad (2)$$

where the task is to find a witness for the first-order variable y . We can also allow y to be a tuple of variables, effectively specifying multiple functions, all depending on the same computable symbols/inputs.

In the rest of the paper, we use specifications like (2), formalized as follows:

Definition 1 (Total Function Synthesis). *Let Σ be a signature, let $u \subseteq \Sigma$ be the set of uncomputable symbols, let $y \in \Sigma \setminus u$, and let $c = \Sigma \setminus (\{y\} \cup u)$. A synthesis problem is a tuple*

$$\langle \Phi[y], u, y \rangle,$$

where $\Phi[y] = \varphi[y, c, u]$ is a quantifier-free formula called a synthesis specification, containing the variable y , which represents the function to be synthesized. A solution for the problem is a term $R \in T_c$, such that $\Phi[R/y]$ holds, called a realizer for y . As R defines a total function, we call this problem total function synthesis.

We consider theory-defined functions computable. In the following we always use y to represent the function to be synthesized, and sometimes write Φ for $\Phi[y]$, and when t does not contain y , we write $\Phi[t]$ for $\Phi[t/y]$.

Example 1 (2-knapsack). We encode the well-known problem: Given 2 items of weights w_1, w_2 and a capacity c , which of the items should we choose so that the total weight of the chosen items is maximal but at most c ? First, we define a formula specifying that y_1, y_2 form a valid pick from the two items, i.e., that the value of y_i is either 0 or w_i (for $i = 1, 2$), and that $y_1 + y_2$ is at most c :

$$\begin{aligned} \varphi[y_1, y_2] := & (y_1 \simeq 0 \vee y_1 \simeq w_1) \wedge (y_2 \simeq 0 \vee y_2 \simeq w_2) \\ & \wedge y_1 + y_2 \leq c \end{aligned}$$

Then the following synthesis problem specifies, assuming that c, w_1, w_2 are non-negative, that y is a valid pick from the two items, such that all valid picks u have lesser or equal combined weight than y :

$$\begin{aligned} & \langle (c \geq 0 \wedge w_1 \geq 0 \wedge w_2 \geq 0) \\ & \implies (\varphi[y] \wedge (\varphi[u] \implies u_1 + u_2 \leq y_1 + y_2)), \{u\}, y \rangle \end{aligned} \quad (3)$$

A realizer for (y_1, y_2) of (3) is:

case $w_1 + w_2 \leq c : (w_1, w_2);$
case $w_1 \leq c \wedge (w_1 \geq w_2 \vee w_2 > c) : (w_1, 0);$
case $w_2 \leq c \wedge (w_2 > w_1 \vee w_1 > c) : (0, w_2);$
default : $(0, 0)$

B. The Expressivity of the Uncomputable Symbols

We define the scope of uncomputable symbols directly in the specification, by using explicit (and possibly second-order) quantification. This is in contrast with [21], which introduced uncomputable symbols, but forbade their use in the sought solution as an additional syntactic constraint, thus allowing the specifications to stay within first-order logic. These two methods of restricting the uncomputable symbols are, however, equivalent in what they allow to express.

The use of uncomputable symbols allows for more expressive specifications, which in turn leads to more interesting synthesis tasks. As in Example 1, with uncomputables we can require that the solution is optimal by changing the specification $\varphi[y, c]$ into $\varphi[y, c] \wedge (\varphi[u, c] \implies \text{is_better}(y, u))$, using a fresh uncomputable symbol u and some suitable definition of the predicate is_better . We can also use uncomputable symbols to ensure that a specification $\varphi[y, c]$ has a solution by adding an assumption $\varphi[u, c]$, allowing to turn partial function specifications into total ones. For example, there is no total solution for the specification $\langle 2y \simeq x, \emptyset, y \rangle$ over integers, but there is one for $\langle 2u \simeq x \implies 2y \simeq x, \{u\}, y \rangle$.¹

Further, u can include also function and predicate symbols. With those, synthesis can require discovering properties of uninterpreted functions or predicates:

Example 2 (Workshop [37], [19]). Consider the synthesis problem $\langle \Phi, \{w\}, y \rangle$, where:

$$\begin{aligned} \Phi := & (Mon \vee Sun) \wedge (Mon \implies w(V)) \wedge (Sun \implies w(A)) \\ & \implies w(y) \end{aligned}$$

The example models a situation at a conference: given that it is Monday or Sunday today, and given that the workshop A is taking place on Sunday and the workshop V on Monday, the task is to synthesize a function that tells us which workshop is taking place. Additionally, the solution is not allowed to use the workshop predicate w – if we were able to evaluate $w(\cdot)$ ourselves, we arguably would not need to use any tools to synthesize the solution.

C. Partial Function Synthesis

We further consider *partial function synthesis*: a variation on the synthesis problem where the specification is not required to be satisfiable on all evaluations of computable symbols. Thus, the sought function is not required to be total.

Definition 2 (Partial Function Synthesis). *Let $\Phi[y]$ be a specification, where $\forall c. \exists y. \forall u. \Phi[y]$ is not necessarily valid. Then $\langle \Phi[y], u, y \rangle$ is a partial (function) synthesis problem. A solution for the problem is a pair $\langle C, R \rangle$, where C is a quantifier-free formula and R is a realizer for y , both over c , such that:*

$$\begin{aligned} C & \implies \Phi[R] & (4) \\ (\exists y. \forall u. \Phi) & \implies C & (5) \end{aligned}$$

The first condition ensures that $\Phi[R]$ holds under C , and the second condition ensures that $\langle C, R \rangle$ is a *maximal solution for partial function synthesis*.

While solutions $\langle C, R \rangle$ are recursively enumerable for first-order specifications, we note that they are in the undecidable $\forall\exists\forall$ -fragment [24]. The process of finding C that satisfies condition (5) relates to synthesizing uniform quantifier-free interpolants [42].

¹This trick does not work if φ already uses some uncomputable symbols. However, our synthesis algorithm (Section III-D) works for partial functions, as defined in Section III-C.

Example 3. Let $\Phi := q(y) \vee a \simeq b \simeq y$ with $u = \{q\}, c = \{a, b\}$. Then the weakest quantifier-free formula that implies $\forall u. \Phi$ is $a \simeq b \simeq y$, producing the partial solution $\langle a \simeq b, a \rangle$. On the other hand, for $\Phi := p(u) \vee a \simeq b \simeq y$, with $u = \{u\}, c = \{p, a, b\}$, the strongest $C \in T_c$ implied by $\forall u. \Phi$ is \top , so while $\langle a \simeq b, a \rangle$ satisfies (4), it does not satisfy (5).

Finite quantifier-free uniform interpolants do not always exist for combinations of theories:

Example 4. Let $\Phi := a \simeq y \wedge \Psi$, where Ψ does not have a uniform quantifier-free interpolant. For example, let Ψ be $\neg(p(u) \wedge a < u < b)$ for integers u, a, b , where $u = \{u\}$. Then there is no finite maximal solution to the synthesis problem, only an infinite one: $\langle \bigvee_{k>1} (a+k \simeq b \wedge \bigwedge_{0<i<k} \neg p(i+a)), a \rangle$.

Thus, we cannot develop complete algorithms in general. Yet, for theories admitting uniform interpolation, or even quantifier elimination, computing the strongest C satisfying (5) is possible. Our synthesis algorithm, presented next, finds partial solutions to (4) and uses (5) as a criterion for termination.

D. A Synthesis Algorithm

We present an algorithm for partial function synthesis. It iteratively builds a list of pairs $R := C_1 \rightarrow r_1 \mid \dots \mid C_n \rightarrow r_n$, stopping when $\forall u. \Phi \wedge \bigwedge_{1 \leq i \leq n} \neg C_i$ becomes unsatisfiable (step 2) – i.e., when the conditions C_1, \dots, C_n cover all possible cases. Then the solution is $\langle \bigvee_{1 \leq i \leq n} C_i, R \rangle$. To compute one pair $C_j \rightarrow r_j$, we first find a formula $\pi^\forall[y]$ not containing symbols from u , which implies $\forall u. \Phi \wedge \bigwedge_{1 \leq i \leq j-1} \neg C_i$ (step 3). This $\pi^\forall[y]$ corresponds to a condition under which the specification holds, yet none of the other conditions found so far apply. Next we look for a witness r and a formula π^\exists , both only containing symbols from c , such that π^\exists implies $\pi^\forall[r]$ (step 4). If we find such r and π^\exists , we set $C_j := \pi^\exists, r_j := r$ and add $C_j \rightarrow r_j$ to the list of pairs R (step 6). Otherwise we find a formula π^\exists which implies $\exists y. \pi^\forall[y]$, and refine the specification by adding $\neg \pi^\exists$ to it (step 5). Then we continue with the next iteration: if $\forall u. \Phi \wedge \bigwedge_{1 \leq i \leq j} \neg C_i$ is satisfiable, we look for $C_{j+1} \rightarrow r_{j+1}$. Computation of π^\forall and π^\exists (steps 3 and 4) is addressed in Section IV.

The algorithm can also be used for total function synthesis by checking that the returned condition C is equivalent to \top .

Correctness of Algorithm 1 is straightforward:

Proposition 3. *Assume Algorithm 1 terminates with $\langle C, R \rangle$. If it never produced $r = \perp$ in step 4, then $\langle C, R \rangle$ is a maximal solution for the given partial synthesis problem. Otherwise $\langle C, R \rangle$ is a program satisfying condition (4) but there is no solution to (5).*

Proof. Each step preserves (4) $C \implies \Phi[R]$. The termination condition ensures (5) $\exists y. \forall u. \Phi \implies C \vee C^\perp$, where C^\perp is the disjunction of projections for $r = \perp$.

Remark 4. *Algorithm 1 is complete for theories that admit quantifier elimination for computing π^\forall, π^\exists in steps 3 and 4.*

Example 5. Our algorithm solves the workshop problem from Example 2. In the first iteration, $\forall w. \Phi$ is sat. We find $\pi^\forall[y] = \neg Mon \wedge \neg Sun$ (see Example 9 for details). Then a corresponding $\langle \pi^\exists, r \rangle$ is $\langle \neg Mon \wedge \neg Sun, _ \rangle$, where $_$ is the

Algorithm 1 Partial Function Synthesis

Given a partial function synthesis task $\langle \Phi, u, y \rangle$, return a solution $\langle C, R \rangle$.

- 1) Initialize $C := \perp$ and $R := \epsilon$.
 - 2) If $\forall u. \Phi \wedge \neg C$ is unsat then return $\langle C, R \rangle$.
 - 3) Find a satisfiable $\pi^\forall[y]$ over $c \cup \{y\}$, such that $\pi^\forall[y] \implies \forall u. \Phi \wedge \neg C$. (See Algorithm 3 in Section IV-B.)
 - 4) Find $\langle \pi^\exists, r \rangle$ such that π^\exists is satisfiable and over c and either $\pi^\exists \implies \pi^\forall[r]$ with $r \in T_c$, or $r = \perp$ and there is no realizer over c for $\pi^\forall[y]$ and $\pi^\exists \implies \exists y. \pi^\forall[y]$. (See Algorithm 2 in Section IV-A.)
 - 5) If $r = \perp$, update $\Phi := \Phi \wedge \neg \pi^\exists$ and go to step 2.
 - 6) Otherwise, update $C := C \vee \pi^\exists$, $R := R \mid (\pi^\exists \rightarrow r)$ and go to step 2.
-

“any” realizer. We add $\neg Mon \wedge \neg Sun$ to C . As $\forall w. \Phi \wedge \neg C$ is satisfiable, we continue. We find $\pi^\forall[y] = A \simeq y \wedge Sun$, and then a corresponding $\langle \pi^\exists, r \rangle = \langle Sun, A \rangle$ (see Example 7). We add $\neg Sun$ to C . The formula $\forall w. \Phi \wedge \neg C$ is still satisfiable, so we continue. In the next iteration, $\pi^\forall = V \simeq y \wedge Mon$ and $\langle \pi^\exists, r \rangle = \langle Mon, V \rangle$. Then finally $\forall w. \Phi \wedge \neg C$ is unsat, and in fact, $C \equiv \top$. We thus return

$$\langle \top, (\neg Mon \wedge \neg Sun \rightarrow _) \mid (Sun \rightarrow A) \mid (Mon \rightarrow V) \rangle.$$

Since the first case in the realizer list uses $_$, we can choose to return V in that case. Further, since its condition $\neg Mon \wedge \neg Sun$ is the complement of the disjunction of the other two conditions (Sun or Mon), we can move it to the end of the list and then unite into a default case with the condition Mon , which also returns V , obtaining the final program:

$$\langle \top, \text{case } Sun : A; \text{default} : V \rangle$$

E. Unique Realizers

A useful special case in step 4 of Algorithm 1 is when a solution to the variable y is uniquely determined. For example, for a simplified 1-item version of knapsack, given by $c < 0 \vee w_1 < 0 \vee ((y_1 = 0 \vee y_1 = w_1) \wedge y_1 \leq c \wedge ((u_1 \neq 0 \wedge u_1 \neq w_1) \vee u_1 > c \vee u \leq y))$, and under the case $c \geq 0, w_1 \geq 0, u_1 = w_1, u_1 > c$, there is a unique solution to y given by 0. In Section V, we show how to read out unique solutions from a combination of the E-graph (a data structure capturing congruences, see Definitions 6 and 7) and arithmetical constraints. Our approach can be considered a generalization of E-graph saturation, where we consider also the theory of linear real arithmetic.

IV. QUANTIFIER PROJECTION

Our synthesis algorithm (Algorithm 1) uses two flavors of quantifier projection methods, one for existential and one for universal quantifiers. Prior work [6] explores projection operators for EUFLIA, but side-steps some requirements we have for the resulting formulas and realizers. We therefore introduce heuristics suitable for our setting. We describe

methods for existential projection in Section IV-A, and Section IV-B presents an approach for universal projection. We then plug these methods into Algorithm 1. For step 3 of Algorithm 1, we use universal projection (Algorithm 3), which takes $\Phi \wedge \neg C$ and returns a set of literals \mathcal{L} , which we assign to π^\forall . For step 4 of Algorithm 1, we use existential projection with witness extraction for the appropriate theory (for EUF, see Algorithm 2), which takes π^\forall and an arbitrary model of it, and returns a suitable pair $\langle \pi^\exists, r \rangle$.

A. Existential Projection

To compute realizers, we use existential model-based projection.

Definition 5 (\exists MBP and \exists MBPR [27]). *Let Φ be a quantifier-free formula and assume $\mathcal{M} \models \Phi$. Then existential model-based projection \exists MBP(\mathcal{M}, x, Φ) is a quantifier-free formula π^\exists over symbols not in x , such that $\mathcal{M} \models \pi^\exists$ and $\pi^\exists \implies \exists x. \Phi$.*

Further, when y is a symbol in Φ , a model-based projection with realizer \exists MBPR(\mathcal{M}, y, Φ) is a pair $\langle \pi^\exists, r \rangle$ such that π^\exists is quantifier-free and satisfiable, $\pi^\exists \implies \Phi[r/y]$, and $y \notin \pi^\exists, r$.

Various algorithms for model-based projection have been integrated into Z3. We will use \exists MBP to project uncomputable symbols in Section IV-B. We also introduce an algorithm for computing \exists MBPR($\mathcal{M}, y, \mathcal{L}$) for EUF given a set of literals \mathcal{L} . Prior works [6], [26] develop projection algorithms that apply to combinations with arrays [25], integer arithmetic, and algebraic data-types [7]. However, they do not cover extracting computable realizers from projections. For example, [6] defines projections of first-order functions that can be defined without getting into notions of computable realizers. On the other hand, we can rely on witness extraction algorithms for LIA [31] for a combination with EUF.

Definition 6 (Congruence Preserving Partition). *Let \mathcal{T} be a set of terms closed under sub-terms. A congruence preserving partition of \mathcal{T} is a set \mathcal{P} that partitions \mathcal{T} , such that for every $f(s), f(t) \in \mathcal{T}$, whenever $\forall i \in \{1, \dots, n\}. \exists P \in \mathcal{P}. s_i, t_i \in P$, then $\exists P' \in \mathcal{P}. f(s), f(t) \in P'$.*

Definition 7 (E-graph, E). *An E-graph, E, is a data structure that provides access to a congruence preserving partition. It maps each term $t \in \mathcal{T}$ to its partition by $\text{root}(t)$.*

Every interpretation \mathcal{M} induces a congruence preserving partition. Conversely, for a set of equalities over EUF, there is a unique congruence preserving partition \mathcal{P} such that for each equality $s \simeq t$, s, t belong to the same class in \mathcal{P} , and every other partition with that property is obtained by taking unions of classes in \mathcal{P} . When a solver constructs an E-graph E, then every equality that is added to E is justified by a set of literals. Justification extends to every implied equality and can be represented as a proof tree that uses axioms for EUF. Thus, when $\text{root}(t) = \text{root}(t')$, then the function $\mathcal{J}(t, t')$ returns the set of equality literals used to establish the equality between t, t' in E. We use the partition to find a computable term $t' \in T_c$ equal to a given term t . To this end, we define a

so-called representative function rep_c , returning a suitable t' for the given t , as well as for the class t belongs to in \mathcal{P} .

Definition 8 ($\text{rep}_c(P)$ [6], [14]). *Given a congruence preserving partition \mathcal{P} , a representative, overloaded as: $\text{rep}_c : 2^{\mathcal{T}} \rightarrow T_c \cup \{\perp\}$, $\text{rep}_c : \mathcal{T} \rightarrow T_c \cup \{\perp\}$, is a function satisfying:*

- 1) $\text{rep}_c(t) = \text{rep}_c(P)$ for each $P \in \mathcal{P}, t \in P$.
- 2) $\text{rep}_c(P) = \perp$ or $\text{rep}_c(P) = f(\text{rep}_c(t_1), \dots, \text{rep}_c(t_n)) \in T_c$ such that $f(t) \in P, \text{rep}_c(t_i) \neq \perp$ for each t_i .
- 3) *Maximality: for every $P \in \mathcal{P}$, $\text{rep}_c(P) = \perp$ implies that every term $f(t)$ in P either has $f \notin c$, or $\text{rep}_c(t_i) = \perp$ for some i .*

Searching for a realizer of y for $\mathcal{L}[y]$ can be performed by Algorithm 2.²

Algorithm 2 $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ for EUF

Given $\mathcal{L}[y]$ over $c \cup \{y\}$, and \mathcal{M} such that $\mathcal{M} \models \mathcal{L}[y]$, return $\langle \pi^\exists, r \rangle$ over c such that either $\pi^\exists \implies \mathcal{L}[r]$, or $r = \perp$ and $\pi^\exists \implies \exists y. \mathcal{L}[y]$.

- 1) Let \mathcal{P} be a congruence preserving partition, based on \mathcal{M} , satisfying \mathcal{L} .
 - 2) Let $\mathcal{C} := \{\text{rep}_c(P) \mid P \in \mathcal{P}, \text{rep}_c(P) \neq \perp\} \cup \{f(t) \mid f \in c, t_i = \text{rep}_c(P_i) \neq \perp\}$.
 - 3) If there is $r \in \mathcal{C}$ such that $\mathcal{L}[y] \wedge y \simeq r$ is satisfiable, return $\langle \mathcal{L}[r], r \rangle$.
 - 4) Otherwise, there is no computable r such that $\mathcal{L}[r]$ does not contradict equalities following from \mathcal{M} . Return $\langle \exists\text{MBP}(\mathcal{M}, y, \mathcal{L}), \perp \rangle$.
-

Proposition 9 (Correctness and Completeness of Algorithm 2). *Algorithm 2 produces $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ iff one exists, and returns $\langle \exists\text{MBP}(\mathcal{M}, y, \mathcal{L}), \perp \rangle$ otherwise.*

Proof. The goal is to establish if there exists a term $r \in T_c$ such that $\mathcal{L}[r] \equiv \mathcal{L}[y] \wedge y \simeq r$ is satisfiable. For this purpose we enumerate a subset of T_c that is necessary and sufficient. In particular, it is sufficient to check the set of terms in \mathcal{P} that are already represented by computable terms, and terms immediately obtained from these terms by using the signature c . Suppose there is a term $r \in T_c$ such that $\mathcal{L}[r]$ is satisfiable. Then either it is congruent to a term in $\text{rep}_c(P), P \in \mathcal{P}$, or we can trim it down to a term with immediate children in $\text{rep}_c(P)$. Suppose $r = f(t)$ is a solution, but t_i for some i is not congruent within \mathcal{P} . Then we could use t_i in place of r because we can produce a model of $\mathcal{L}[t_i]$ by setting the interpretation of t_i the same as $f(t)$. Note that it is possible that \mathcal{M} enforces congruences not in \mathcal{P} so it is not necessarily the case that $\mathcal{M} \models \mathcal{L}[r]$.

Remark 10. *Single invocations of $\exists\text{MBPR}$ do not necessarily produce formulas that are equivalent to $\exists y. \mathcal{L}[y]$. Thus, $\mathcal{L}[y] \wedge \neg \mathcal{L}[r]$ is not necessarily unsatisfiable. However, $\exists\text{MBPR}$ for EUF+LIA finitely eliminates all models of \mathcal{L} .*

Example 6. In the following constraints, there are no solutions for y using T_c . (1) Let $c = \{a\}$ and $\mathcal{L} = y \not\approx a$. (2) Let $c = \{f, a\}$ and $\mathcal{L} = f(a) \simeq a \wedge y \not\approx a$.

²We describe the practical optimizations for Algorithm 2, including finding the “any” realizer \perp , in Section VI.

Example 7. We compute $\langle \pi^\exists, r \rangle$ for $\mathcal{L}[y] = \{A \simeq y, \text{Sun}\}$, requested in Example 5. A partition \mathcal{P} based on \mathcal{L} is $\{\{A, y\}, \{\text{Sun}, \top\}\}$. Then $\mathcal{C} = \{A, \text{Sun}\}$. Since A is of the suitable sort and $\mathcal{L}[A] \wedge y \simeq A$ is satisfiable, the result is $\langle \text{Sun} \wedge A \simeq A, A \rangle$, equivalent to $\langle \text{Sun}, A \rangle$.

Extending MBP from LIA to EUFLIA requires enforcing Ackermann axioms. We derive a recipe from [6].

Definition 11 ($\exists\text{MBPR}$ for LIA + EUF). *Let $\mathcal{M} \models \mathcal{L}$ over LIA + EUF, and assume \mathcal{L} is closed under Ackermann reductions with respect to y . That is, for every occurrence $f(s)$ and $f(t)$, where f is uninterpreted, s, t are arithmetic terms that contain y : either $(s \not\approx t) \in \mathcal{L}$ or $(f(s) \simeq f(t)) \in \mathcal{L}$. Then $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ is defined by LIA projection where only occurrences of y within arithmetic terms are considered.*

Remark 12. *We established complete $\exists\text{MBP}$ and $\exists\text{MBPR}$ methods for EUF and EUFLIA. $\exists\text{MBP}$ for combinations of theories that include arrays and algebraic datatypes are implemented in Z3.*

B. Universal Projection

Universal projection for EUFLRA was addressed in different guises. Covering algorithms by [16] can be used to compute projections for EUFLRA. They provide a strongest Ψ such that $(\exists u. \neg \Phi) \implies \Psi$. Equivalently, $\neg \Psi \implies \forall u. \Phi$. We would like to unfold $\neg \Phi$ incrementally as a DNF so that we can compute implicants of $\forall u. \Phi$ incrementally and terminate once we find a suitable projection.

We here outline a method that produces a DNF of implicants. It can be used for ground formulas and combinations of theories where $\exists\text{MBP}$ applies. It derives from the MARCO algorithm [32] for core enumeration, where our twist is to apply projection to cores during enumeration and to terminate early when a suitable satisfying assignment is found. Let Φ be a ground formula and \mathcal{L} be all the literals that occur in the negation normal form of $\neg \Phi$. The set of cores over \mathcal{L} with respect to Φ can be enumerated using core enumeration techniques. Each core \mathcal{C} is such that $\mathcal{C} \wedge \Phi$ is unsatisfiable, which means $\Phi \implies \neg \mathcal{C}$. We can filter out cores that are unsatisfiable by themselves. It remains to eliminate u from each core, or in other words, find \mathcal{C}' , such that $(\forall u. \neg \mathcal{C}) \implies \neg \mathcal{C}'$. This question is equivalent to finding \mathcal{C}' , such that $\mathcal{C}' \implies \exists u. \mathcal{C}$. We can use $\exists\text{MBP}$ to establish \mathcal{C}' based on a model for \mathcal{C} . Thus,

$$\begin{aligned} \forall u. \Phi &\equiv \forall u. (\Phi \wedge \bigwedge_i \neg \mathcal{C}_i) \\ &\equiv (\forall u. \Phi) \wedge \bigwedge_i \forall u. \neg \mathcal{C}_i \\ &\equiv (\forall u. \Phi) \wedge \bigwedge_i \neg \mathcal{C}'_i \end{aligned}$$

We have found a sufficient set of cores when (if) the formula

$$\left(\bigwedge_i \neg \mathcal{C}'_i \right) \wedge \neg \forall u. \Phi$$

is unsat. But we do not need to compute the full set of cores before we can extract candidates for π^\forall . We can return a propositional model π^\forall of the current partial set of projections $\bigwedge_i \neg \mathcal{C}'_i$, whenever $\pi^\forall \wedge \neg \forall u. \Phi$ is unsat. In other words, it

suffices to extract a DNF of $\bigwedge_i \neg \mathcal{C}'_i$ on demand without having the full $\bigwedge_i \neg \mathcal{C}'_i \implies \forall \mathbf{u}. \Phi$.

This discussion motivates Algorithm 3. It enumerates propositional models of literals in Φ , using additional constraints \mathcal{O} (blocking implicants of $\neg\Phi$) and \mathcal{S} (blocking implicants of Φ which use uncomputable symbols), both initialized to \top . The models are constructed by first finding a subset U of uncomputable literals in $\neg\Phi$ consistent with $\mathcal{O} \wedge \mathcal{S}$, and then finding a model for $U \wedge \Phi$. If this model satisfies a set of literals \mathcal{L}_c^\top not using uncomputables, and $\mathcal{L}_c^\top \implies \Phi$, we are done (step 7). If $U \wedge \Phi$ is unsat, we block this model in future enumeration: we project \mathbf{u} from the unsat core $\text{Core}(U, \Phi)$, negate the projection obtaining $\bigvee_{\ell \in \bar{\mathcal{C}}} \ell$ (corresponding to $\neg \mathcal{C}'_i$ from the discussion above), add this clause to \mathcal{O} , and add the new literals from $\bar{\mathcal{C}}$ to all clauses in \mathcal{S} (step 4). The last case is when the propositional model implies Φ , but using an implicant which includes uncomputable literals. To block this model in future enumeration iterations, we add a clause over literals that are false in this model to \mathcal{S} (step 8). In contrast to MARCO, the set of literals that is used to enumerate cores grows (in step 4). To allow models using the new literals, the algorithm weakens \mathcal{S} by including new literals from $\bar{\mathcal{C}}$. The algorithm is resumable after returning in step 7 by supplying a clause $\bigvee_{\ell \in \bar{\mathcal{C}}} \ell$ implied by $\neg \pi^\forall$ to step 4(a).

Algorithm 3 Universal projection using cores, correction sets and $\exists\text{MBP}$

Given Φ and \mathbf{u} , return π^\forall such that either $\pi^\forall \implies \forall \mathbf{u}. \Phi$, or $\pi^\forall = \perp$.

- 1) Let $\mathcal{O} := \top, \mathcal{S} := \top$, let \mathcal{L}_u be literals in $\text{NNF}(\neg\Phi)$ that contain symbols from \mathbf{u} , and let \mathcal{L}_c be the literals in $\text{NNF}(\Phi)$ that do not contain symbols from \mathbf{u} .
 - 2) If $\mathcal{O} \wedge \mathcal{S}$ is unsat, then return \perp .
 - 3) Find a subset $U \subseteq \mathcal{L}_u \cup \mathcal{L}_c$ that is satisfiable with model \mathcal{M} and implies $\mathcal{O} \wedge \mathcal{S}$.
 - 4) If $U \wedge \Phi$ is unsat, let $\mathcal{C} := \text{Core}(U, \Phi)$, let $\bar{\mathcal{C}} := \{\bar{\ell} \mid \ell \in \exists\text{MBP}(\mathcal{M}, \mathbf{u}, \mathcal{C})\}$, and
 - a) add $\bigvee_{\ell \in \bar{\mathcal{C}}} \ell$ to \mathcal{O} , update $\mathcal{S} := \{S \vee \bigvee_{\ell \in (\bar{\mathcal{C}} \setminus \mathcal{L}_c)} \ell \mid S \in \mathcal{S}\}$, $\mathcal{L}_c := \mathcal{L}_c \cup \bar{\mathcal{C}}$, and
 - b) go to step 2.
 - 5) Otherwise, let \mathcal{M}' be such that $\mathcal{M}' \models U \wedge \Phi$.
 - 6) Let $\mathcal{L}_c^\top := \{\ell \in \mathcal{L}_c \mid \mathcal{M}'(\ell) = \top\}$
 - 7) If $\mathcal{L}_c^\top \wedge \neg\Phi$ is unsat, then \mathcal{L}_c^\top is an implicant of Φ . Return $\pi^\forall := \text{Core}(\mathcal{L}_c^\top, \neg\Phi)$.
 - 8) Let $\mathcal{L}^\perp := \{\ell \in \mathcal{L}_c \cup \mathcal{L}_u \mid \mathcal{M}'(\ell) = \perp\}$, add $\bigvee_{\ell \in \mathcal{L}^\perp} \ell$ to \mathcal{S} , and go to step 2.
-

Proposition 13 (Correctness of Algorithm 3). *If Algorithm 3 returns $\pi^\forall \neq \perp$, then $\pi^\forall \implies \forall \mathbf{u}. \Phi$.*

Proof. Since the algorithm can only return π^\forall from step 7, it means $\pi^\forall = \text{Core}(\mathcal{L}_c^\top, \neg\Phi)$ where \mathcal{L}_c^\top does not contain symbols from \mathbf{u} , and $\pi^\forall \implies \Phi$, which then entails $\pi^\forall \implies \forall \mathbf{u}. \Phi$.

Remark 14. *The algorithm provides a way to incrementally compute cases of a partial function. It is complete for theories that admit quantifier elimination.*

Proof. Claim: the sets \mathcal{L}_c obtained from the NNF of Φ and \mathcal{L}_u obtained from the NNF of $\neg\Phi$ are sufficient for enumerating conflicts with Φ , and $\neg\Phi$, respectively. If L is a propositional model such that $L \wedge \Phi$ is unsat then $L' := L \setminus \{\ell \in L \mid \ell \notin \mathcal{L}_u, \bar{\ell} \in \mathcal{L}_u\}$ also satisfies $L' \wedge \Phi$ is unsat. The symmetric claim is: If L is a propositional model such that $L \wedge \neg\Phi$ is unsat, then $L' := L \setminus \{\ell \in L \mid \ell \notin \mathcal{L}_c, \bar{\ell} \in \mathcal{L}_c\}$ also satisfies $L' \wedge \neg\Phi$ is unsat. Justification (for the last claim): If $\bar{\ell}$ occurs positive in Φ , but ℓ does not occur in Φ at all, then ℓ does not occur negatively in $\neg\Phi$ and therefore doesn't contribute to $L \wedge \neg\Phi$ being unsat.

Claim: All propositional models of $\bigwedge_i \neg \mathcal{C}'_i$ are covered by step 7. Here, \mathcal{C}'_i is a projection of \mathcal{C}_i and \mathcal{C}_i is a propositional model of Φ . It follows as step 4 prunes literal combinations that are propositional models of $\neg\Phi$, while step 7 identifies the models for $\bigwedge_i \neg \mathcal{C}'_i$, and step 8 forces enumeration of propositional models to contain literals that belong to a correction set of non-models of Φ .

Put together, if the underlying theory allows to finitely eliminate the propositional models (in step 4 by quantifier elimination), we eventually reach the model corresponding to the implicant of $\forall \mathbf{u}. \Phi$.

Example 8. Let $\Phi := f(u) \simeq a \implies f(u) \simeq y$ where $\mathbf{c} = \{f, a\}$, $\mathbf{u} = \{u\}$. Let $U := \{f(u) \simeq a, f(u) \not\simeq y\}$. Then $U \wedge \Phi$ is unsat and the core $\mathcal{C} := f(u) \simeq a \wedge f(u) \not\simeq y$ with respect to Φ has the projection $a \not\simeq y$. We add $a \simeq y$ to \mathcal{O} , and now $\mathcal{O} \wedge \neg\Phi$ is unsatisfiable and we can return $a \simeq y$ in step 7. Existential projection computes the pair $\langle \top, a \rangle$, which is a solution to the synthesis problem.

Example 9. We compute π^\forall requested by Algorithm 1 in Example 5. We have $\Phi := ((\text{Mon} \vee \text{Sun}) \wedge (\text{Mon} \implies w(V)) \wedge (\text{Sun} \implies w(A))) \implies w(y)$ and $\mathbf{u} = \{w\}$. First let $U := \{\neg \text{Mon}, \text{Sun}, w(A), w(V), \neg w(y)\}$. Then $U \wedge \Phi$ is unsat, with $\mathcal{C} = \text{Core}(U, \Phi) = \{\text{Sun}, w(A), \neg w(y)\}$, projected to $\{\text{Sun}, A \not\simeq y\}$. We let $\mathcal{O} := \neg \text{Sun} \vee A \simeq y$ and find a new $U := \{\neg \text{Mon}, \neg \text{Sun}, w(A), w(V), \neg w(y)\}$. Then $U \wedge \Phi$ is sat, and we find its model $\mathcal{M}' = \{\text{Sun} \mapsto \perp, \text{Mon} \mapsto \perp, A \mapsto v_0, V \mapsto v_1, y \mapsto v_2, w \mapsto \lambda v.v \neq v_2\}$. We get $\mathcal{L}_c^\top = \{\neg \text{Mon}, \neg \text{Sun}\}$, which implies Φ , and we thus return $\pi^\forall = \text{Core}(\mathcal{L}_c^\top, \neg\Phi) = \{\neg \text{Mon}, \neg \text{Sun}\}$.

V. SYNTHESIZING UNIQUE FUNCTIONS

When specifications have unique realizers, we can take important shortcuts. For synthesis tasks based on Definition 2, there may be unique solutions to y based on implicants π^\forall of Φ . In that case, we can bypass invoking $\exists\text{MBPR}$. It is well-recognized in QBF solving [36] that there can be a substantial advantage to detecting when a formula entails that there is a unique solution for an existential variable. Heuristics based on unique solutions are also used to significantly aid synthesis of Boolean functions [1]. We here outline how we leverage unique function specifications in the case of EUF and LRA.

The setting we will be discussing is the following: given a variable y , we introduce inferences that can establish when a set of constraints Φ entails that y is equal to a term t that does not contain any of the symbols from $\{y\} \cup \mathbf{u}$. For LRA in isolation, there is a canonical method for inferring equalities from conjunctions of equalities and inequalities: Gaussian elimination. For EUF in isolation, we can consult rep_c from Definition 8 if the congruence closure induced by a conjunction of equalities implies that y is congruent to a term without symbols from $\{y\} \cup \mathbf{u}$. We summarize these methods in a bit more detail next. Our main observation is that the *combination* of EUF and LRA is also amenable to synthesizing unique solutions. In Section V-C we show how to extract unique solutions for y (if they exist) in a combination of the two theories. The main claim is that Gaussian elimination can be interleaved with congruence closure rules to establish all equalities modulo LRA + EUF under a set of asserted literals. It builds on an idea of using Gaussian elimination to solve for symbols that should be eliminated and use the resulting equations to augment the congruence closure for equalities that may have not been surfaced by the LRA solver. Recall, that it is enough for theory combinations to agree on equalities with respect to a candidate model [12] and not all implied equalities. So our method extracts implied equalities on the fly from LRA and takes them into account by augmenting the congruence closure. Beyond experimenting with this method on our benchmarks, this approach also lets Z3 expose equality saturation modulo EUF and LRA.

A. EUF

The heart of CDCL(T) solvers is a congruence graph E for the set of terms in Φ . In a satisfiable state, E maintains a set of currently implied congruences. If in a satisfiable state $\text{rep}_c(y) \in T_c$ and $\mathcal{J}(y, \text{rep}_c(y))$ has no terms from \mathbf{u} , then we have a justification that y is equal to a computable term using assumptions that are computable or use y . Assuming the state is consistent for EUF, the justification $\mathcal{J}(y, \text{rep}_c(y))$ is also EUF-consistent. We then produce the condition $\mathcal{J}(y, \text{rep}_c(y))[\text{rep}_c(y)/y]$ and the realizer $\text{rep}_c(y)$. By construction, this approach extracts realizers that are forced by E , but it may fail to detect that a set of constraints have only one possible realizer:

Example 10. Let $\Sigma = \{a, b\}$, $\mathcal{L} := y \neq b \wedge a \neq b$. Then a is a unique realizer for y , but for the congruence class over \mathcal{L} , $\text{rep}_c(y) = \perp$.

B. LRA

It is also possible to solve for variables over LRA. We recall the dual tableau normal form used by the arithmetic solver. A tableau T maintains a set of basic, \mathcal{B} , and non-basic, \mathcal{N} , variables such that each basic variable is in a solved form from the non-basic variables. The sets \mathcal{B} and \mathcal{N} are disjoint.

$$T : \bigwedge_{b \in \mathcal{B}} \left(x_b = \sum_{i \in \mathcal{N}} a_{bi} x_i \right)$$

Furthermore, it maintains bounds B on all variables $\bigwedge_i \text{lo}(x_i) \leq x_i \leq \text{hi}(x_i)$, where lower $\text{lo}(x_i) \in \{-\infty\} \cup \mathbb{R}$,

$\text{hi}(x_i) \in \{\infty\} \cup \mathbb{R}$. Every lower and upper bound that is not infinite is justified by a set of literals $\mathcal{J}_{\text{lo}}(x)$, $\mathcal{J}_{\text{hi}}(x)$. A feasible tableau comes with an evaluation function $\text{val}(x_i)$ that maps every variable into a value from \mathbb{R} , such that:

$$\text{val}(x_b) = \sum_{i \in \mathcal{N}} a_{bi} \text{val}(x_i) \quad \text{for every } b \in \mathcal{B}$$

Let x be an arithmetic variable. It can be reduced to a solution in terms of other variables or constants in the following ways:

- 1) $x = \text{lo}(x)$, when x is fixed by existing bounds $\text{lo}(x) = \text{hi}(x)$.
- 2) $x = \text{val}(x)$, when the current value for x cannot be changed, i.e., when $(x > \text{val}(x) \vee x < \text{val}(x)) \wedge T \wedge B$ is infeasible.
- 3) $x = \sum_i a_i x_i$, when $x \in T$ and $T', x = \sum_i a_i x_i$ is obtained from T by pivoting x to a base variable.

C. EUF + LRA

Suppose there is a term $t \in T_c$ such that \mathcal{L} entails $y \simeq t$. Based on the ingredients for EUF and LRA we establish a proof search for t . Represent \mathcal{L} as $E + T + B$, then the search succeeds by deriving the sequent $E, T, B \vdash y \rightsquigarrow t$ with $t \in T_c$, using the following rules:

$$\begin{array}{c} \frac{E, T, B \vdash t_i \rightsquigarrow s_i \text{ for each } i}{E, T, B \vdash \sum_i a_i t_i \rightsquigarrow \sum_i a_i s_i} \\[10pt] \frac{\text{rep}_c(t) \neq \perp}{E, T, B \vdash t \rightsquigarrow \text{rep}_c(t)} \quad \frac{t \text{ is fixed by } T, B}{E, T, B \vdash t \rightsquigarrow \text{val}(t)} \\[10pt] \text{Congruence} \frac{E \vdash t = f(\mathbf{t}), f \in \mathbf{c} \quad E, T, B \vdash t_i \rightsquigarrow t'_i \text{ for each } i}{E, T, B \vdash t \rightsquigarrow f(\mathbf{t}')} \\[10pt] \text{Gauss} \frac{T \equiv t = \sum_i a_i t_i, T' \quad E, T', B \vdash t \rightsquigarrow t'}{E, T, B \vdash \sum_i a_i t_i \rightsquigarrow t'} \end{array}$$

Proposition 15. *For a real-valued variable y , the search for $t \in T_c$ with $E, T, B \vdash y \rightsquigarrow t$ saturates finitely and finds a solution in T_c if and only if it exists.*

Proof (Outline). The proof search is finite because every time it solves using T it removes the row for the solved variable through a Gauss elimination step, and the number of terms in E to check is finite. Assuming arithmetic operations are computable, the rule that decomposes summations is a special case of the congruence rule. The proof search is also complete for arithmetical LRA since if it is invoked with y and finitely saturates without producing a term for y , we can establish a model where y is distinct from every term in T_c : Set $\text{val}(y) := \text{val}(y) \pm \epsilon$ for an infinitesimal ϵ , and inductively, for extensions $t = \sum_i a_i t_i$ there is at least one term t_i that cannot be solved for, so set $\text{val}(t_i) := \text{val}(t_i) \pm \epsilon/a_i$. The sign $\pm \epsilon$ is chosen to make sure the new value of y is within the bounds for y in B . Whenever t_i is congruent to $f(\mathbf{t})$, $f \in \mathbf{c}$, there is inductively an argument t_i which is non-solvable, so the interpretation of f

can be adjusted to take into account the new value for t_i . This produces a non-standard model where the interpretation of all terms in T_c are over the rational numbers while y evaluates to a non-standard real. Functions are interpreted over rationals if all arguments are rationals, and can otherwise evaluate to a non-standard real if some argument is not a real number.

We note that it is straightforward to extend the inference system to also extract justifications by tracking inequalities used for fixed bounds and equalities used for congruences.

VI. IMPLEMENTATION AND EVALUATION

We implemented our method as a proof of concept, and compared it with state-of-the-art methods supporting uncomputables: the synthesis mode of Vampire [21] and the leading SyGuS solver cvc5 [39], obtaining encouraging early results.

A. Implementation

We implemented Algorithms 1, 2, and 3, as well as the unique solution finding of Section V, as a stand-alone Python prototype (available online at [20]) using the Z3 API, which we also extended by new features. We use variables of array sort for higher-order quantification. The theory of arrays in Z3 coincides with SMT-LIB 2 [5] for ground formulas, but not quantified formulas. E.g., the first-order theory of array select function admits models where select does not include bijections, but with Z3 one can instantiate quantified array variables by $\lambda x.x$ and select is just a function application.

The API already provided functions for computing $\exists\text{MBP}$, an incomplete method for $\exists\text{MBPR}$, and accessing the E-graph. We extended Z3 by exposing E-graph justifications, and by a new heuristic for finding witnesses from T_c for EUF in $\exists\text{MBPR}$. It searches for an E-match for y that does not lead to equating shared sub-terms, and specifically avoids merging sub-terms used in disequalities. We also added a method for solving variables that is complete for LRA and extends to LIA (incompletely, not to deal with non-convexity of LIA).

We optimize all our algorithms as follows. In Algorithm 1, step 6, if $u = \emptyset$, we use $\Phi[r/y]$ as the condition for realizer r instead of π^\exists . Further, to compute $\exists\text{MBPR}(\mathcal{M}, y, \mathcal{L})$ of Algorithm 2, we first check if \mathcal{L} contains y , and if it does not, we return the “any” realizer (as in Example 5). Otherwise, we try calling $\exists\text{MBPR}$ of the API, and if that does not return a realizer, we iterate over the candidates provided by rep_c , only producing a new one if the last did not work out. Finally, in Algorithm 3, step 7, we check the unsatisfiability with respect to $\neg\Phi'$, where Φ' is (i) initialized to the original specification Φ , and (ii) only if Algorithm 3 returns the same π^\forall twice, updated to $\Phi' := \Phi \wedge \neg C$ using the current C of Algorithm 1.

B. Evaluation

To the best of our knowledge, there was no standard benchmark set for synthesis with uncomputable symbols as of writing this paper. However, a dataset based on the benchmarks from [21] was under development in parallel with this paper,

Id	Benchmark subset description	Count	Has u	Partial	Unique
I	simple if-then-else benchmarks from SyGuS competition [3]	3	✗	✗	✓
II	k -knapsack (see Example 1)	5	✓	✗	✓
III	maximum function for n given integers	10	✗	✗	✓
IV	lower bound function for n integers	10	✗	✗	✗
V	function finding an integer lower than all n given integers	10	✗	✗	✗
VI	function finding an integer between the given bounds, <u>assuming</u> some exists	4	✓	✗	✗
VII	function finding an integer between the given bounds, <u>if</u> some exists	4	✓	✓	✗
VIII	function finding a solution for y in the given equation, assuming it exists	3	✓	✗	✓
IX	function finding a solution for y in the given equation, <u>if</u> it exists	3	✗	✓	✓
X	Examples 2 (and a computable variant thereof) and 8 (and a variant thereof)	4	✗+✓	✗	✓
XI	the first specification or Example 3	1	✓	✓	✓

TABLE I: Summary of the 57 benchmarks used: set id, description, how many benchmarks are in the subset, and whether they use uncomputables, specify a partial function, and a unique function. Benchmark sets I-IX use LIA, X and XI use UF. In set II we have $k \in \{1, 2, 3, 4, 5\}$, and in sets III-V we have $n \in \{2, 5, 10, 15, 20, 25, 30, 50, 100, 150\}$.

	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	sum	total & partial
Synthesiz3	3	3	9	4	3	4	4	3	3	4	1	33	& 8
Vampire	3	0	5	2	0	4	—	0	—	2	—	16	& —
cvc5	3	1	5	5	5	4	—	0	—	4	—	27	& —

TABLE II: Experimental results: numbers of problems solved by each solver in benchmark sets I-XI. Benchmark sets VII, IX, and XI specify partial synthesis problems not supported by Vampire nor cvc5, which we denote by “—”.

and in the meantime it was accepted for publication [18].³ We took the subset of this dataset supported by our method,⁴ and extended it with the solvable examples from this paper, as well as additional partial function synthesis problems. Table I shows a summary of the benchmarks, organized into 11 subsets denoted I-XI. The set contains 57 benchmarks: 18 small, conceptually distinct problems (all 3 problems of subset I, 2 per both VI and VII, 4 in X, and one per all the other subsets), and 39 scaled-up versions of the small instances.

We evaluated the performance of our prototype on this dataset and for the total synthesis problems also compared

³The dataset was co-developed by one of the authors of this paper. It is available at https://github.com/vprover/vampire_benchmarks/tree/master/synthesis. The version that was available shortly before this paper was submitted, and which we used for evaluation, is at https://github.com/vprover/vampire_benchmarks/tree/6356e52/synthesis.

⁴We excluded problems using (i) theories beyond EUFLIRA, or (ii) more complex quantification, or (iii) interpreted uncomputables, or (iv) which were too similar to others in the dataset – e.g. we used maximum but not minimum.

it to Vampire and cvc5.⁵ We ran each benchmark on a single core of a 1.60GHz CPU with 16 GB RAM, with the time limit of 1 minute. We chose 1 minute as it is enough time to solve the solvable small problems, and show the trend for the scaled-up problem instances, but not solve all the scaled-up instances. Table II shows the numbers of solved problems. Successful solving of the small instances took all solvers less than 1 second. For the larger instances, the runtimes increased up to the time limit. We note that what makes many scaled-up instances hard is that they encode problems with many symmetries: e.g., each case of the maximum-of- n -integers function has the same structure, but with different variables.

Overall, our approach solves the most total function synthesis problems, and also partial synthesis problems which cvc5 and Vampire do not support. Our approach scales as well as or better than both Vampire and cvc5 for all benchmarks with a unique solution, and slightly worse than cvc5 on problems with infinitely many possible solutions (sets IV and V). Notably, within 1 minute, only our approach solves 2- and 3-knapsack, and it also finds the maximum-of- n -integers function for n up to 100, while cvc5 and Vampire can go only up to $n \leq 20$. Interestingly, both cvc5 and Vampire fail to solve all benchmarks of set VIII, including the smallest instance $\langle 2u \simeq x \implies 2y \simeq x, \{u\}, y \rangle$, “find the half of an even x ”.

VII. RELATED WORK

Synthesis: The field of synthesis encompasses very different approaches. We only overview methods similar to ours: synthesizing functions from scratch based on specifications given as logical formulas with some syntactic restrictions. The general restriction setting in [40] is to integrate a black-box function within a solver that determines whether a term is computable. We base our specification format on [21]. In contrast with [21], we distinguish uncomputable symbols directly in the specification formula by using quantification. The deductive synthesis methods of [21], [40] are based on theorem proving, and [21] is implemented in Vampire [29]. While these methods introduce specialized inference rules to filter terms using uncomputable symbols, our approach uses quantifier elimination games to remove uncomputables, and thus requires no changes in the underlying search procedures.

Another related paradigm is SyGuS [2], which supports specifying multiple functions with different inputs, and constraining the space of solutions to functions generated by a given grammar. SyGuS solvers include cvc5 [4], [39] and DryadSynth [22]. Compared to the specification setting we consider, SyGuS allows a more fine-grained syntactic control over the synthesized function. We note that to solve such specifications, we could generate candidate terms using the provided grammar instead of using \exists MBPR, or that we

could use grammar-based reconstruction as in [39] on top of \exists MBPR. On the other hand, our specifications support the use of uninterpreted symbols, in particular uninterpreted functions and predicates. This allows us to express e.g. the workshop problem (see Example 2), which is not directly supported by SyGuS. However, using higher-order logic, uninterpreted functions can be encoded as higher-order variables, and with this encoding cvc5 solves the workshop problem (and some other problems, see Section VI). Finally, neither SyGuS nor the deductive synthesis methods support partial function synthesis.

We provide some examples where the partial function synthesis problem has no solutions. The study of unrealizability [34] dually establishes methods to show that there are no solutions to synthesis problems.

Projections: The literature on interpolation, including [17], develops algorithms based on proof objects. Symbol elimination in saturation [28] works by deriving clauses that are consequences of $\exists u. \neg \Phi$. Negations of these clauses can be used as projections. The approach of [28] relies on symbol orderings to eliminate symbols through rewriting, but does not offer guarantees of complete symbol elimination. Deductive synthesis with uncomputables [21] blocks inferences which would lead to uncomputable realizers, but also is not complete. As an example, inferences such as $f(y) \not\simeq f(a) \vdash y \not\simeq a$ with $u = \{f\}$ are not part of this methodology.⁶

Notably, MCSAT [23] relies on a dual to \exists MBP, called \forall MBP, in a setting where it is used for satisfiability checking of quantifier-free formulas. The starting point for \forall MBP is a partial model of a subset of symbols that cannot be completed to a full model of all symbols. A combination of \forall MBP and \exists MBP for quantifier solving for polynomial arithmetic is developed in [35], while \exists MBP with cores and strategies are used in [7], [9], [33]

We note that stand-alone projection is not necessarily the most efficient method for solving quantifiers. Templates have recently been used successfully for non-linear arithmetic [11], and Z3’s Horn clause solver uses information from several projection invocations to compute more general projections [30].

Implied equalities: Bromberger and Weidenbach [10] present a method for extracting all implied equalities from a set of linear inequalities over the reals. It is used when solving over LIA. Z3 [8] uses an incomplete heuristic to detect implied equalities during search. The QEL algorithm [14] uses equalities that are consistent with an interpretation, but not necessarily implied by a current assignment to literals, to simplify quantifier elimination problems.

VIII. CONCLUSIONS

This paper formally defines and studies the problem of synthesis of total and partial functions in the presence of uncomputable symbols. To solve these problems, we introduced a modular algorithm for synthesis, relying on quantifier projections. We summarized requirements on the projections

⁵We used Vampire v4.9, commit dde88d57f, in the configuration `--decode dis+32_1:1_tgt=off:qa=synthesis:ep=off:alasc a=off:drc=ordering:bd=off:nm=0:sos=on:ss=included:si=on:rawr=on:rtra=on:proof=off:mss=off_600`, and cvc5 v1.1.2 in the default configuration. For cvc5, we used SyGuS translations of the problems, encoding uncomputables as non-input variables.

⁶We note that such an inference could be supported by a suitable variation of unification with abstraction [38].

and presented algorithms computing existential projection with witnesses for EUFLIA, and universal projection internally relying on existential projection. Further, we described a method to circumvent quantifier solving for uniquely defined functions in EUFLRA. Finally, we implemented a prototype demonstrating that the proposed methods work in practice.

Our results prompt future work in multiple directions, such as extending synthesis to functions with different inputs, or extending the specification with quantified background axioms to support reasoning with user-defined theories. Moreover, developing projections for other theories would allow synthesis in those theories by plugging them into our algorithm. Last but not least, our unique solution approach could be extended to infer all implied equalities.

Acknowledgments. We thank Kuldeep Meel for suggesting to examine partial function synthesis with unique realizers. Diego Olivier Fernandez Pons inspired us to explore synthesis with Z3. We also thank Mikoláš Janota for discussions of examples. Petra Hozzová was funded by the European Union under the project ROBOPROX (reg. no. CZ.02.01.01/00/22_008/0004590).

REFERENCES

- [1] S. Akshay, Supratik Chakraborty, and Shetal Shah. Tractable representations for boolean functional synthesis. *Ann. Math. Artif. Intell.*, 92(5):1051–1096, 2024.
- [2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emmina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.
- [3] Rajeev Alur, Dana Fisman, Saswat Padhi, Andrew Reynolds, Rishabh Singh, and Abhishek Udupa. SyGuS-Comp 2019. <https://sygus.org/comp/2019/>, 2019.
- [4] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proc. of TACAS*, pages 415–442, 2022.
- [5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [6] Nikolaj Bjørner, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Model based interpolation for uninterpreted functions and integer linear arithmetic. EasyChair Preprint no. 10000, EasyChair, 2023.
- [7] Nikolaj S. Bjørner and Mikoláš Janota. Playing with quantified satisfiability. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.
- [8] Nikolaj S. Bjørner and Lev Nachmanson. Arithmetic solving in Z3. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 26–41. Springer, 2024.
- [9] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Christophe Vauthier. QSMA: A new algorithm for quantified satisfiability modulo theory and assignment. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2023.
- [10] Martin Bromberger and Christoph Weidenbach. Fast cube tests for LIA constraint solving. In *IJCAR*, 2016.
- [11] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Mehrdad Karrabi, Harshit J Motwani, Maximilian Seeliger, and Đorđe Žikelić. Quantified Linear and Polynomial Arithmetic Satisfiability via Template-based Skolemization. In *AAAI*, 2025.
- [12] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, May 2008.
- [13] Azadeh Farzan and Zachary Kincaid. Linear arithmetic satisfiability via strategy improvement. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 735–743. IJCAI/AAAI Press, 2016.
- [14] Isabel Garcia-Contreras, Hari Govind V. K., Sharon Shoham, and Arie Gurfinkel. Fast approximations of quantifier elimination. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 64–86. Springer, 2023.
- [15] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-Free Programs. In *Proc. of PLDI*, page 62–73, 2011.
- [16] Sumit Gulwani and Madan Musuvathi. Cover algorithms and their combination. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2008.
- [17] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In Hongseok Yang, editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2011.
- [18] Márton Hajdu, Petra Hozzová, Laura Kovács, Andrei Voronkov, Eva Maria Wagner, and Richard Steven Žilínčík. Synthesis Benchmarks for Automated Reasoning. <https://arxiv.org/abs/2507.19827>, 2025. To appear in proceedings of CICM 2025.
- [19] Petra Hozzová. Integrating Answer Literals with AVATAR for Program Synthesis. In Laura Kovács and Michael Rawson, editors, *Proc. of the 7th and 8th Vampire Workshop*, volume 99 of *EPiC Series in Computing*, pages 13–20. EasyChair, 2024.
- [20] Petra Hozzová and Nikolaj Bjørner. Online material for Synthesiz3 This! <https://doi.org/10.5281/zenodo.16436706> and <https://github.com/Z3Prover/doc/tree/master/synthesiz3>, 2025.
- [21] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. In Brigitte Pientka and Cesare Tinelli, editors, *Proc. of CADE*, volume 14132 of *LNCS*, pages 307–324, Cham, 2023. Springer.
- [22] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling Enumerative and Deductive Program Synthesis. In *Proc. of PLDI*, PLDI 2020, page 1159–1174, New York, NY, USA, 2020. Association for Computing Machinery.
- [23] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.
- [24] A. S. Kahr, Edward F. Moore, and Hao Wang. Entscheidungsproblem reduced to the $\forall\exists\forall$ case. *Journal of Symbolic Logic*, 27(2):225–225, 1962.
- [25] Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *FMCAD*, pages 89–96, 09 2015.
- [26] Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design, FMCAD '15*, page 89–96, Austin, Texas, 2015. FMCAD Inc.
- [27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.

- [28] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2009.
- [29] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *Proc. of CAV*, volume 8044 of *LNCs*, pages 1–35. Springer, 2013.
- [30] Hari Govind Vadiramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. *Formal Methods Syst. Des.*, 63(1):81–109, 2024.
- [31] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In Benjamin G. Zorn and Alex Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 316–329. ACM, 2010.
- [32] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.
- [33] Charlie Murphy and Zachary Kincaid. Quantified linear arithmetic satisfiability via fine-grained strategy improvement. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 89–109. Springer, 2024.
- [34] Shaan Nagy, Jinwoo Kim, Thomas W. Reps, and Loris D’Antoni. Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. *Proc. ACM Program. Lang.*, 8(OOPSLA2):113–139, 2024.
- [35] Jasper Nalbach and Gereon Kremer. Extensions of the cylindrical algebraic covering method for quantifiers. *CoRR*, abs/2411.03070, 2024.
- [36] Markus N. Rabe, Leander Tentrup, Cameron Rasmussen, and Sanjit A. Seshia. Understanding and extending incremental determinization for 2qbf. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2018.
- [37] Giles Reger. Revisiting Question Answering in Vampire. *EPiC Series in Computing*, 53:64–74, 2018.
- [38] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning. In *Proc. of TACAS*, pages 3–22, 2018.
- [39] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.*, 55(2):73–102, 2019.
- [40] Tanel Tammet. Completeness of resolution for definite answers. *J. Log. Comput.*, 5(4):449–471, 1995.
- [41] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program Synthesis Using Dual Interpretation. In *Proc. of CADE*, pages 482–497, 2015.
- [42] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In Robert Nieuwenhuis, editor, *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.