EPiC Series in Computing

Volume 100, 2024, Pages 484–502

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Automated Synthesis of Decision Lists for Polynomial Specifications over Integers

S. Akshay[1], Supratik Chakraborty[1], Amir Kafshdar Goharshady[2], R. Govind[3], Harshit Jitendra Motwani[2], and Sai Teja Varanasi[1]

[1] IIT Bombay, India, {akshayss, supratik}@cse.iitb.ac.in, 200050152@iitb.ac.in
[2] HKUST, Hong Kong, {goharshady, csemotwani}@ust.hk
[3] Uppsala University, Sweden, govind.rajanbabu@it.uu.se

Abstract

In this work, we consider two sets $\mathbb{I}$ and $\mathbb{O}$ of bounded integer variables, modeling the inputs and outputs of a program. Given a specification Post, which is a Boolean combination of linear or polynomial inequalities with real coefficients over $\mathbb{I} \cup \mathbb{O}$, our goal is to synthesize the weakest possible pre-condition Pre and a program $\mathbf{P}$ satisfying the Hoare triple $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$. We provide a novel, practical, sound and complete algorithm, inspired by Farkas' Lemma and Handelman's Theorem, that synthesizes both the program $\mathbf{P}$ and the pre-condition Pre over a bounded integral region. Our approach is exact and guaranteed to find the weakest pre-condition. Moreover, it always synthesizes both $\mathbf{P}$ and Pre as linear decision lists. Thus, our output consists of simple programs and pre-conditions that facilitate further static analysis. We also provide experimental results over benchmarks showcasing the real-world applicability of our approach and considerable performance gains over the state-of-the-art.[1]

1 Introduction

Automated program synthesis is often considered one of the holy grails of Computer Science. Not surprisingly, research on this topic has had a long history (e.g., see [34]), and significant advances have been reported in recent years [5, 7, 36, 50, 31]. Informally, the synthesis problem can be described using two sets of variables, say $\mathbb{I}$ and $\mathbb{O}$, representing program inputs and outputs respectively, and a logical formula $\varphi(\mathbb{I}, \mathbb{O})$ specifying a desired relation between values of $\mathbb{I}$ and $\mathbb{O}$. The synthesis task is to generate a program $\mathbf{P}$ that accepts values for $\mathbb{I}$ and generates values for $\mathbb{O}$ such that $\varphi(\mathbb{I}, \mathbb{O})$ is satisfied. If the specification is given in the form $\mathsf{Pre} \Rightarrow \mathsf{Post}$, where Pre is a (pre-condition) predicate over $\mathbb{I}$ and Post is a (post-condition) predicate over $\mathbb{I} \cup \mathbb{O}$, the synthesizer must generate a terminating program $\mathbf{P}$ such that the Hoare triple $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$ is satisfied. Oftentimes, however, it is much easier for a user to give the specification simply as a post-condition Post over $\mathbb{I} \cup \mathbb{O}$, and require the synthesizer to generate both a pre-condition Pre and a terminating program $\mathbf{P}$ such that $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$ is satisfied. As shown in Section 2, given Post, it may be impossible in general to find a program

[1] A longer version, including appendices, is available at [4].

that computes values of $\mathbb{O}$ for every value of $\mathbb{I}$ such that Post is satisfied. Hence, identifying an appropriate pre-condition Pre is extremely important in practice. At the same time, a trivial but useless solution can always be obtained by setting Pre to false. To disallow such degenerate solutions, we require the synthesized pre-condition Pre to be the (logically) weakest. This, in turn, implies that the program $\mathbf{P}$ must work for all and only those values of $\mathbb{I}$ for which there exists some value of $\mathbb{O}$ that satisfies $\varphi(\mathbb{I}, \mathbb{O})$. For such values, the program can create any output as long as it satisfies the specification, i.e. our goal is not to find all satisfying outputs. This flavour of automated synthesis, which is our focus in this paper, has been studied earlier in the context of functional synthesis (see e.g. [41, 3, 43]). However, these earlier works either restricted the specification to linear constraints over rational/integer variables, or restricted the variables to be of Boolean type. Our interest in this paper is to go beyond these restrictions and allow polynomial constraints for post-conditions.

The problem of (weakest) pre-condition and program synthesis is, computationally hard and even infeasible in most practical settings. For instance, if we consider $\mathbb{I}$ and $\mathbb{O}$ to assume unbounded integer values, and if we allow Post to include polynomial inequalities over $\mathbb{I}$ and $\mathbb{O}$, synthesizing the weakest pre-condition and program becomes undecidable [13], thanks to the Matiyasevich-Robinson-Davis-Putnam theorem [26]. In finite domains, there have been several related works on synthesizing Skolem functions, such as [45, 39, 27]. Since pre-condition synthesis reduces to quantifier elimination, this particular problem is computable if $\mathbb{I}$ and $\mathbb{O}$ assume real values, due to Tarski's seminal result [52, 49]. Nevertheless, implementations based on quantifier elimination techniques like cylindrical algebraic decomposition (CAD) [23, 24] result in significant numerical precision issues, and are often impractical. In other words, even if Post has polynomial inequalities with real coefficients, we would like the $\mathbb{I}$ and $\mathbb{O}$ variables to take only rational (or integral) values, so that they can be represented efficiently and computed precisely. Since computations with rational or integral values on a finite-precision computer boils down to computations with bounded integers, we focus on the weakest pre-condition and program synthesis problem with bounded integers. While the problem becomes trivially computable in this setting, it is not immediately clear whether we can do any better than a naive enumeration of integral points within the hypercube defined by the integer bounds. Such an approach would be infeasible in most cases, as the number of integral points in the space defined by the constraints is often exponentially large. Surprisingly, we show in Section 3 that the worst-case exponential complexity cannot be circumvented unless well-regarded complexity theoretic conjectures are falsified. Nevertheless, as our work shows, in many practical settings, this worst-case behaviour can be circumvented by a careful design of algorithms.

Specifically, in this paper, we take a step towards developing practically efficient algorithms for weakest pre-condition and program synthesis for specifications given as (Boolean combinations of) polynomial constraints over a bounded integral domain. In doing so, we are first faced with the question of representation and nature of the program synthesized. While for several applications, any synthesized program in any format suffices, for further use and evaluation, we often need to synthesize a program that is easy to represent and allows for further optimization. In this work, we propose to represent (a) pre-conditions as decision lists that use bound checks on input variables as decision predicates, and (b) programs as *valuation lists* that mimic decision lists in their structure, but return integral values of outputs instead of Boolean values of decisions. Interestingly, several state-of-the-art synthesis tools such as CVC5 [9] often generate programs that are deeply nested `if-then-else` structures, which can be viewed as valuation lists. Decision-lists [46] are of course a well-known formalism for representing Boolean functions, and are known to be very useful for representation and analysis in the Boolean functional synthesis context (see e.g., [14]). In this work, we aim to adapt them to the synthesis of general

integer programs.

In summary, given Post as a Boolean combination of linear and/or polynomial inequalities over $\mathbb{I} \cup \mathbb{O}$, we present a technique to synthesize the weakest pre-condition Pre and a program $\mathbf{P}$ as decision (and value) lists, such that the Hoare triple $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$ holds over bounded integral domains for $\mathbb{I}$ and $\mathbb{O}$. We accomplish this by using a combination of specialized techniques from polyhedral and real algebraic geometry, i.e. Farkas' Lemma and Handelman's Theorem. Our contributions can be listed as follows:

- We show that the problem of computing the weakest pre-condition even over bounded integral domains cannot avoid an exponential enumeration in the worst case, unless the exponential-time hypothesis (ETH) is falsified. ETH [38], is a well-regarded hypothesis in computational complexity theory that is widely considered to hold. Hence, our result shows that it is extremely unlikely that sub-exponential algorithms for weakest pre-condition and program synthesis exist.
- We provide a novel algorithm based on mesh refinement and results from polyhedral and algebraic geometry, that synthesizes both the program $\mathbf{P}$ and the weakest pre-condition Pre over a bounded integral region. We show soundness and completess of the algorithm, and argue why it avoids exponential enumeration in practice. When the specification is given as linear constraints over inputs and outputs, we adapt the classical Farkas' Lemma [28] from polyhedral geometry, while for the case of non-linear constraints, we are inspired by Handelman's Theorem [35] from real algebraic geometry.
- We show that our algorithms always synthesize weakest pre-conditions Pre as decision lists, and programs $\mathbf{P}$ as valuation lists, that are inspired by decision lists. Thus, our output consists of programs and pre-conditions in a simple representation that facilitate further optimizations and enhancements. For instance, it is easy to generalize the valuation lists to provide a range of output values (instead of a single output value) for each input value.
- Finally, we provide experimental results over benchmarks showcasing the applicability and effectiveness of our approach.

Related Work. For integer variables, prior work has considered synthesizing programs starting from a grammar in the Syntax Guided Synthesis (SyGuS) approach [5, 6, 29, 7]. The state-of-the-art competition-winning SyGuS tools, such as CVC5 [9] and DryadSynth [36], can synthesize programs if either the pre-condition is true, or if it is known and provided as an assumption to the tool, even over non-linear constraints. However, none of these tools can compute the weakest pre-condition, and indeed the SyGus approach is not geared towards this. The approach used by most SyGuS tools is also search based and often uses the syntactic template (i.e., the grammar) to enumerate over possible programs. A related line of work also considers synthesizing programs by filling holes in user-provided sketches or templates [50, 31, 44]. This approach has also been applied to termination and runtime analysis of programs [15, 17, 16, 37, 10], invariant generation [18, 47, 25, 42, 30], static cost analysis of probabilistic programs [21, 19, 54, 53, 22, 51], detection of deep bugs [8], and LTL model-checking [20]. A more recent but quite orthogonal paradigm is example-driven synthesis, where programs are synthesized from examples using techniques from formal methods and machine learning, e.g., [48, 11, 33]. Other approaches include reducing program synthesis to the problem finding a witness of a dependency quantified formula modulo theory [32].

On the other hand, synthesizing pre-conditions along with programs over integral and non-integral domains has been studied in [41], where the authors propose a procedure that synthesizes the weakest pre-condition and the corresponding program. In contrast to our procedure,

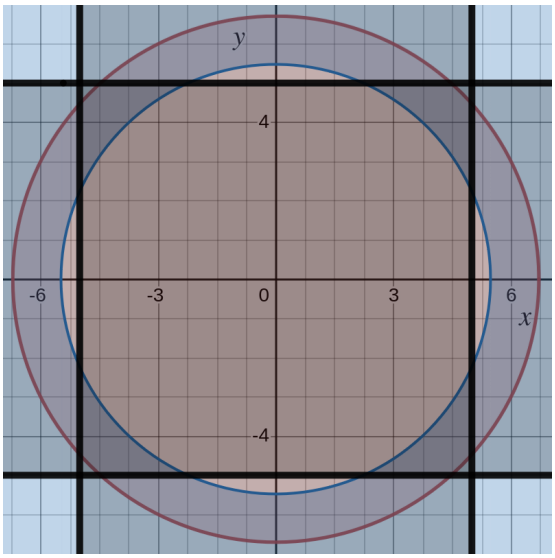

Figure 1: Illustrating specification for pre-condition + program synthesis

the procedure in [41] is not restricted to a bounded domain; however, their procedure focuses on linear integer arithmetic and does not extend to non-linear constraints. Regarding representation of programs as decision lists, we wish to highlight that other than Boolean setting where this is common (e.g.,[14]), the programs given out by tools such as CVC5 are often decision lists as well. However, significant improvement is possible in this direction to obtain more succinct representations such as linear decision diagrams [12] .

The structure of the paper is as follows. We start with a motivating example in Section 2. In Section 3, we formalize the problem statement, show theoretical hardness and develop our synthesis algorithm. Section 4 is devoted to our experimental results and we end with a Conclusion in Section 5.

2 Motivating Example

In this section, we illustrate the need for jointly synthesizing weakest pre-conditions and programs over integers, starting from non-linear specifications. Suppose we wish to synthesize a program that takes as input an integer representing the x-coordinate of a point in the 2-dimensional plane, and returns an integer for the y-coordinate such that the resulting point lies within the region defined by $\varphi(x, y) = (x^2 + y^2 \geq 30) \wedge (x^2 + y^2 \leq 45)$. Suppose further that x and y are both constrained to lie between -5 and 5, both inclusive. Thus, the points (x, y) that satisfy the specification are those in the dark grey annular corners in Fig. 1.

Now, there exist values of x, for which it is impossible to find (integral) values of y such that $\varphi(x, y)$ holds. For example, if $x = 2$, there is no value of y in $[-5, 5]$ that satisfies $\varphi(x, y)$. This motivates the need for computing a weakest pre-condition predicate $\psi(x)$, such that the input x satisfies $\psi(x)$ iff there is an integer value of y in $[-5, 5]$ that satisfies $\varphi(x, y)$. Such a pre-condition can be used to filter out input values for which it is futile to try to generate outputs that satisfy the given specification. For our example, $\psi(x)$ is $(-5 \leq x \leq -3) \vee (3 \leq x \leq 5)$.

Once $\psi(x)$ is obtained, we need a program that calculates a value of y for each x satisfying $\psi(x)$ such that $\varphi(x, y)$ holds. In general, there may be many programs that serve this purpose. For example, both programs `Pa` and `Pb` shown below and written in C-like syntax, work for our example. While we will be mostly concerned with synthesizing any one such program, we discuss later how the user may be given the choice of which output value to use.

```
if  −5 ≤ x ≤ −5 then return y ↦ 4
if  −4 ≤ x ≤ −3 then return y ↦ 5
if  3 ≤ x ≤ 4 then return y ↦ 5
if  5 ≤ x ≤ 5 then return y ↦ 4
else return fail
```

Program (Pa)

```
if  −5 ≤ x ≤ −4 then return y ↦ −4
if  −3 ≤ x ≤ −3 then return y ↦ −5
if  3 ≤ x ≤ 3 then return y ↦ 5
if  4 ≤ x ≤ 5 then return y ↦ 4
else return fail
```

Program (Pb)

It is worth noting that state-of-the-art program synthesis tools like CVC5 [9] report the specification $\varphi(x, y)$ for our example as "infeasible", without outputting any program. This is because the weakest pre-condition $\psi(x)$ is not a tautology. Hence such tools cannot be directly used to synthesize the (weakest pre-condition, program) pair for a given specification. In contrast, the tool developed in this paper outputs the pre-condition $\psi(x)$ discussed above, and also generates one of the programs shown above in less than a second (see Example 16 in Table 1). Interestingly, when we feed the pre-condition generated by our tool as an additional input to CVC5, it generates an `if-then-else` program with non-linear conditions in the `if` statements. In contrast, programs generated by our tool always check bounds of input variables in `if` statements, as in `Pa` and `Pb`.

3 Our Synthesis Algorithm

In this section, we first formalize our problem and then provide our synthesis algorithm, which is inspired by, but not dependent on, two well-known theorems in polyhedral and real algebraic geometry, namely Farkas' Lemma and Handelman's Theorem.

3.1 Problem Definition

Consider a finite set $\mathbb{V} = \mathbb{I} \cup \mathbb{O}$ of bounded integer-valued variables. We assume that we have bounds $L, U \in \mathbb{Z}$ such that each variable can take values in the range $[L, U]$. We also assume that $\mathbb{I}$ and $\mathbb{O}$ are disjoint and call them the *input* and *output* variables respectively. The input variables are read-only and their values cannot be changed by the program[2].

Valuations. A *real valuation* is a function $val^{\mathbb{V}}_{\mathbb{R}} : \mathbb{V} \to [L, U]$, assigning a real value to each variable. Similarly, an *integer valuation* is a function $val^{\mathbb{V}}_{\mathbb{Z}} : \mathbb{V} \to [L, U] \cap \mathbb{Z}$. In this work, we assume that all of our program variables take integer values and thus focus mainly on integer valuations. We also sometimes refer to valuations as points in $\mathbb{R}^{\mathbb{V}}$ or $\mathbb{Z}^{\mathbb{V}}$.

Input. In the input, we are given a specification Post generated from the terminal φ in the grammar below:

$$\begin{array}{lll} \varphi & := & \ell \mid \varphi \wedge \varphi \mid \neg\varphi \\ \ell & := & f \bowtie 0, \text{ where } \bowtie \in \{>, \geq\}, \quad f \in \mathbb{R}[\mathbb{V}] \end{array} \tag{1}$$

[2]We are not considering a separate set for the temporary variables that are created by a program but do not form part of the output. As we will see further below, this is without loss of generality.

Here, f is a polynomial over $\mathbb{V}$, ℓ a *literal* and φ a *formula*. We also often use $\varphi_1 \vee \varphi_2$ as syntactic sugar for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$. In other words, our specification Post is a Boolean combination of polynomial inequalities over the variables.

Output. Our goal is to synthesize a program $\mathbf{P}$ and a weakest pre-condition Pre, such that the Hoare triple $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$ holds. Formally, let $\Pi := \{\mathsf{Pre}' \mid \exists \mathbf{P}' \;\; \{\mathsf{Pre}'\}\mathbf{P}'\{\mathsf{Post}\}\}$ be the set of all possible pre-conditions. We are looking for a pre-condition Pre that satisfies $\mathsf{Pre}' \Rightarrow \mathsf{Pre}$ for all $\mathsf{Pre}' \in \Pi$. Note that in the definition of Π, we have no constraints over the programs and pre-conditions, i.e. $\mathbf{P}'$ is not necessarily a polynomial program and Pre' can be any Boolean combination of predicates over input variables.

Output Format for Pre-conditions. Our algorithm synthesizes Pre as a linear decision list [46] generated by the grammar given below. Note that the guards in this decision list only check bounds on input variables.

$$\begin{array}{lcc} \textsc{Dl} & := & \texttt{if } \varphi^\star \texttt{ then return true ; } \textsc{Dl} \\ & \mid & \texttt{if } \varphi^\star \texttt{ then return true else return false} \\ \varphi^\star & := & \ell^\star \mid \ell^\star \wedge \varphi^\star \\ \ell^\star & := & a \le v \le b, \text{ where } a, b \in \mathbb{Z}, \;\; v \in \mathbb{I} \end{array} \tag{2}$$

In the grammar above v is a variable. Note that the pre-condition Pre may only contain input variables in $\mathbb{I}$. As is standard [46], the semantics of a decision list $\textsc{Dl}$ of the form

$$\begin{array}{l} \texttt{if } \varphi_1^\star \texttt{ then return true} \\ \qquad\vdots \\ \texttt{if } \varphi_n^\star \texttt{ then return true} \\ \texttt{else return false} \end{array}$$

is simply defined as $\varphi_1^\star \vee \cdots \vee \varphi_n^\star$.

Output Format for Programs. Our algorithm not only synthesizes a pre-condition Pre, but also a program $\mathbf{P}$ satisfying the Hoare triple $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$. Our programs $\mathbf{P}$ are also quite simple and generated by the following grammar. The programs in this grammar are similar to decision lists, but return valuations for the output variables. Thus, we call them *valuation lists*.

$$\begin{array}{lcc} \textsc{Vl} & := & \texttt{if } \varphi^\star \texttt{ then return } val_{\mathbb{Z}}^{\mathbb{O}} \texttt{ ; } \textsc{Vl} \\ & \mid & \texttt{if } \varphi^\star \texttt{ then return } val_{\mathbb{Z}}^{\mathbb{O}} \texttt{ else return fail} \\ \varphi^\star & := & \ell^\star \mid \ell^\star \wedge \varphi^\star \\ \ell^\star & := & a \le v \le b, \text{ where } a, b \in \mathbb{Z}, \;\; v \in \mathbb{I} \\ val_{\mathbb{Z}}^{\mathbb{O}} & := & \mathbb{O} \to [L, U] \cap \mathbb{Z} \end{array} \tag{3}$$

Example. Consider the example in Section 2, where $\mathsf{Post} := (x^2 + y^2 \le 45) \wedge (x^2 + y^2 \ge 30)$. The following pre-condition Pre (left) and program $\mathbf{P}$ (right) constitute a valid solution to this problem instance. As discussed in Section 2, Pre is indeed the weakest pre-condition for the given Post; additionally, $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$ holds.

```
if  −5 ≤ x ≤ −4 then return true
if  −3 ≤ x ≤ −3 then return true
if  3 ≤ x ≤ 3 then return true
if  4 ≤ x ≤ 5 then return true
else return false
```

```
if  −5 ≤ x ≤ −4 then return y ↦ −4
if  −3 ≤ x ≤ −3 then return y ↦ −5
if  3 ≤ x ≤ 3 then return y ↦ 5
if  4 ≤ x ≤ 5 then return y ↦ 4
else return fail
```

3.2 Synthesis as Projection

Projection of a Valuation. Let val be a valuation over the program variables $\mathbb{V}$. We define $\pi(val) : \mathbb{I} \to [L, U]$ by simply setting $\pi(val)(v) = val(v)$ for every $v \in \mathbb{I}$. In other words, $\pi(val)$ is a valuation over only the input variables, which is obtained by ignoring the values that val assigns to other variables.

Model Sets. Recall that our specification Post is a Boolean combination of polynomial inequalities over the program variables $\mathbb{V}$. Let $SAT_{\mathbb{R}}(\mathsf{Post}) = \{val_{\mathbb{R}} \mid val_{\mathbb{R}} \models \mathsf{Post}\}$ and $SAT_{\mathbb{Z}}(\mathsf{Post}) = \{val_{\mathbb{Z}} \mid val_{\mathbb{Z}} \models \mathsf{Post}\}$ be the sets of real and integer valuations that satisfy the post-condition Post. By definition, $SAT_{\mathbb{R}}(\mathsf{Post})$ is a semi-algebraic set and $SAT_{\mathbb{Z}}(\mathsf{Post}) = SAT_{\mathbb{R}}(\mathsf{Post}) \cap \mathbb{Z}^{\mathbb{V}}$.

Weakest Pre-condition. Let Pre be a weakest pre-condition for the post-condition Post. Since our programs cannot change the values of their input variables, it follows from the above definitions that $SAT_{\mathbb{Z}}(\mathsf{Pre}) = \pi\left(SAT_{\mathbb{Z}}(\mathsf{Post})\right)$.

Synthesis as Projection. Based on the equation above, the problem of synthesizing Pre is equivalent to finding a decision list that accepts precisely the points in $\pi\left(SAT_{\mathbb{Z}}(\mathsf{Post})\right) = \pi(SAT_{\mathbb{R}}(\mathsf{Post}) \cap \mathbb{Z}^{\mathbb{V}})$. In other words, we are looking for all points that can be obtained as projections of the integral points in a bounded semi-algebraic set.

3.3 Complexity

Given the formulation above, a naïve exponential-time solution would be to just enumerate all the integer points in the semi-algebraic set $SAT_{\mathbb{R}}(\mathsf{Post})$. This is possible since the set is bounded, i.e. each variable takes values between L and U. This proves that our problem is decidable. However, it would lead to a huge decision list / program with one branch for each integral point.

In this section, we provide a conditional hardness result, showing that one cannot hope for a sub-exponential solution unless the exponential time hypothesis (ETH) is false. Thus, if ETH is true, as it is widely believed to be, then it rules out the possibility of sub-exponential algorithms for our problem.

Hypothesis 1 (Exponential Time Hypothesis [38])**.** *Satisfiability of 3-CNF SAT formulas cannot be decided in sub-exponential time* $2^{o(n)}$.

Our Reduction. We provide a reduction from 3-CNF SAT to our problem. Let

$$\phi = (a_{1,1} \vee a_{1,2} \vee a_{1,3}) \wedge (a_{2,1} \vee a_{2,2} \vee a_{2,3}) \wedge \cdots \wedge (a_{m,1} \vee a_{m,2} \vee a_{m,3})$$

be a SAT formula over the Boolean variables $x_1, x_2, \ldots, x_n$ where each literal $a_{i,j}$ is either an x_k or its negation $\neg x_k$. We first obtain a new formula ϕ' by disjuncting the literal x_0 to each clause in ϕ, where x_0 is a new Boolean variable. Note that ϕ' is trivially satisfiable if x_0 is set to true (or 1), while ϕ' is satisfiable with x_0 set to false (or 0) iff ϕ is satisfiable. We now define a polynomial $f(\phi') \in \mathbb{R}[x_0, x_1, \ldots, x_n]$ corresponding to ϕ' using the following inductive construction:

- $f(x_k) = x_k$
- $f(\neg x_k) = 1 - x_k$
- $f(a_1 \vee a_2 \vee a_3) = 1 - (1 - f(a_1)) \cdot (1 - f(a_2)) \cdot (1 - f(a_3))$
- $f(c_1 \wedge c_2) = f(c_1) \cdot f(c_2)$

Let $val : \{x_0, x_1, x_2, \ldots, x_n\} \to \{0, 1\}$ be a Boolean valuation for $x_0, \ldots, x_n$. It is easy to verify that $f(\phi')(val) = 1 \Leftrightarrow val \models \phi'$.

Next, we construct a synthesis problem, where $\mathbb{I} = \{x_0\}$ and $\mathbb{O} = \{x_1, x_2, \ldots, x_n\}$ and $\mathsf{Post} := f(\phi') \geq 1$. Let $\mathsf{Pre}(x_0)$ be the weakest pre-condition we wish to synthesize. From the construction of ϕ', it is clear that $(x_0 = 1) \Rightarrow \mathsf{Pre}$. However, $(x_0 = 0) \Rightarrow \mathsf{Pre}$ holds iff ϕ is satisfiable. Thus, the formula ϕ is satisfiable if and only if $\mathsf{Pre} \Leftrightarrow (x_0 = 0) \vee (x_0 = 1)$. Since x_0 can only be either 0 or 1, we can check if $\mathsf{Pre} \Leftrightarrow (x_0 = 0) \vee (x_0 = 1)$ simply by evaluating Pre for $x_0 = 0$ and for $x_0 = 1$. A sub-exponential algorithm for computing Pre can only generate a pre-condition expression that is at most sub-exponential sized. Assuming that evaluating an expression for given values of variables takes time no more than polynomial in the size of the expression, a sub-exponential algorithm for computing Pre would effectively give us a sub-exponential time algorithm for checking if ϕ is satisfiable. This contradicts the Exponential Time Hypothesis, and we have the following conditional complexity result.

Theorem 1. *Assuming ETH, there is no sub-exponential time algorithm that, given a specification* Post *as a Boolean combination of polynomial inequalities over* $\mathbb{V}$*, finds the weakest pre-condition* Pre *for which there exists a program* $\mathbf{P}$ *such that* $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$.

We note that our reduction actually proves a stronger result. In our reduction, Post is a single polynomial inequality. Moreover, it is not only impossible to find the weakest pre-condition in sub-exponential time but also to decide whether the weakest pre-condition evaluates to true for a given value of inputs. We also remark that the same reduction establishes NP-hardness, too.

Since exponential complexity is likely unavoidable, our main contribution is providing a more practical synthesis algorithm that creates decision lists with fewer branches in practice.

3.4 Mesh Refinement Algorithm

In this section, we present a procedure that is central to our algorithm. Intuitively, this procedure creates a mesh of hypercube cells around the semi-algebraic set $S_{\mathbb{R}} := SAT_{\mathbb{R}}(\mathsf{Post})$. Then, for every cell in the mesh, it tries to decide (i) whether the cell is entirely inside $S_{\mathbb{R}}$ or (ii) whether the cell is entirely outside $S_{\mathbb{R}}$. In the former case, we know that all integral points in the cell are in $S_{\mathbb{Z}} := SAT_{\mathbb{Z}}(\mathsf{Post})$. Similarly, in the latter case, none of the integral points in the cell are in $S_{\mathbb{Z}}$. Finally, if none of the two checks (i) and (ii) pass, we will refine our mesh by subdividing the cell. This continues until the cell contains only one integral point p at which point we can simply check whether $p \models \mathsf{Post}$. Using the mesh, we can easily synthesize a decision list that accepts exactly the points in $S_{\mathbb{Z}}$.

Intuition. Informally, our goal is to avoid enumerating every integral point. We achieve this by trying to decide whether each cell, which can contain exponentially many integral points, is entirely inside or outside the solution. If we succeed, we would not have to enumerate over the integral points of this cell, potentially saving a lot in the runtime. Although this cannot avoid the overall exponential complexity due to Theorem 1, our experimental results in Section 4 show that it makes the approach substantially more scalable in practice. We now formalize our procedure.

Cells. Let $\mathbb{V} = \{v_1, v_2, \ldots, v_n\}$. A *cell* $C \subseteq \mathbb{R}^{\mathbb{V}}$ is the set of points that satisfy inequalities of the form

$$\psi_C := \begin{cases} a_1 \leq v_1 \leq b_1 \\ a_2 \leq v_2 \leq b_2 \\ \quad \vdots \\ a_n \leq v_n \leq b_n \end{cases},$$

where the a_i and b_i's are integer constants and we have $b_i \geq a_i$ for every $1 \leq i \leq n$. In other words, a cell is basically a hypercube with integral bounds. The *diameter* of C is defined as

$\max_{i=1}^{n}(b_i - a_i)$ and the *longest side* of C is $\arg\max_{i=1}^{n}(b_i - a_i)$. We remark that a cell can easily be represented as one branch in our decision lists. More specifically, C can be encoded by a branch of the form

$$\texttt{if } \wedge_{i=1}^{n} a_i \leq v_i \leq b_i \texttt{ then return true} .$$

Mesh. A *mesh* M is a set of finitely many pairwise disjoint cells[3]. Formally $M = \{C_1, C_2, \ldots, C_M\}$ where each C_i is a cell and $C_i \cap C_j = \emptyset$ for all $i \neq j$.

Refinement. Consider a cell C with longest side i. If $a_i = b_i$, then C has exactly one integral point in it. Otherwise, we define the refinement of C as two new cells C_1 and C_2 obtained by cutting C in half along v_i. Note that we keep our bounds integral. Thus, using the same ψ_C as above, we formally have:

$$\psi_{C_1} := \begin{cases} a_1 \leq v_1 \leq b_1 \\ \vdots \\ a_{i-1} \leq v_{i-1} \leq b_{i-1} \\ a_i \leq v_i \leq \lfloor \frac{a_i+b_i}{2} \rfloor \\ a_{i+1} \leq v_{i+1} \leq b_{i+1} \\ \vdots \\ a_n \leq v_n \leq b_n \end{cases} \quad \text{and} \quad \psi_{C_2} := \begin{cases} a_1 \leq v_1 \leq b_1 \\ \vdots \\ a_{i-1} \leq v_{i-1} \leq b_{i-1} \\ \lceil \frac{a_i+b_i}{2} \rceil \leq v_i \leq b_i \\ a_{i+1} \leq v_{i+1} \leq b_{i+1} \\ \vdots \\ a_n \leq v_n \leq b_n \end{cases} .$$

We remark that $C_1 \cup C_2$ is not necessarily equal to C, but every integral point in C is either in C_1 or C_2. Thus, our refinement does not miss any integral points and it is safe to replace C with C_1 and C_2. We extend our definition of refinements to meshes. We say that M' is a refinement of M if it can be obtained by applying a finite number of such replacements, each time taking a cell $C \in M$, removing it and instead adding its refinement cells C_1 and C_2.

Algorithm 1 Mesh Refinement

```
1:  procedure MESH_REFINEMENT(𝕍 = {v_1, v_2, ..., v_n}, Post, L, U)
2:      C_0 ← {val ∈ ℝ^𝕍 | ∀i  L ≤ val(v_i) ≤ U}        ▷ We know that S_ℤ ⊆ S_ℝ ⊆ C_0
3:      Red ← {C_0}
4:      Green ← ∅
5:      while Red ≠ ∅ do
6:          Choose C ∈ Red
7:          Red ← Red \ {C}
8:          if Is_Subset(C, Post, 𝕍) then        ▷ If this check returns true then C ⊆ S_ℝ
9:              Green ← Green ∪ {C}
10:         else if Is_Subset(C, ¬Post, 𝕍) then        ▷ C does not intersect S_ℝ
11:             continue
12:         else if Diameter(C) = 0 then
13:             Choose the unique point p ∈ C
14:             if p ⊨ Post then
15:                 Green ← Green ∪ {C}
16:         else
17:             C_1, C_2 ← Refinement(C)
18:             Red ← Red ∪ {C_1, C_2}
        return Green
```

Our Mesh Refinement Procedure. Algorithm 1 shows our mesh refinement procedure. We start with a single-cell mesh M that contains our whole space and keep refining it. Additionally,

[3]Our cells can intersect in their boundary.

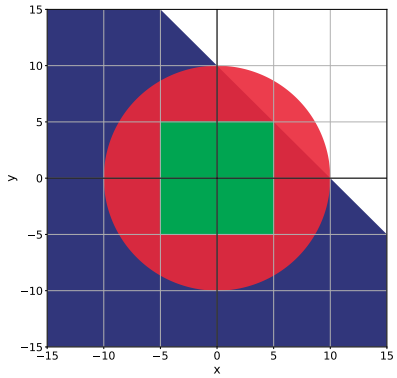

Figure 2: Figure demonstrating Case 1 and Case 2 of our subset checking procedure. The green-colored cell $C := \{-5 \leq x \leq 5, -5 \leq y \leq 5\}$ in the figure is a subset of the region defined by the hyperplane $x + y \leq 10$ and the circle $x^2 + y^2 \leq 100$.

we color each cell in M either red or green. Intuitively, a cell C is red if we are not sure about it. It becomes green as soon as we establish that $C \subseteq S_{\mathbb{R}}$. Additionally, if we find out that $C \cap S_{\mathbb{R}} = \emptyset$, we discard C from our mesh. Initially, the only cell in our mesh is red. At every iteration (line 5), we take an arbitrary red cell C. We then try to decide whether C is a subset of $S_{\mathbb{R}} = SAT_{\mathbb{R}}(\mathsf{Post})$. This is the check in line 8. If it is, then we color C green. Otherwise, we check to see if we can prove that $C \cap S_{\mathbb{R}} = \emptyset$ (line 10). If this check passes, we simply discard C. Finally, if we fail to prove that C is either entirely inside $S_{\mathbb{R}}$ or entirely outside of it, then we refine C (lines 16-18). We continue our refinements until C contains only a single integer point p (line 12). If this happens, we check whether $p \models \mathsf{Post}$. If so, we color C green. The procedure ends when there are no remaining red cells and returns the resulting mesh.

Theorem 2 (Soundness and Completeness)**.** *Suppose we have a sound but not necessarily complete implementation of* Is_Subset(), *i.e. an oracle which returns false if the subset relation is not satisfied, but is not guaranteed to return true when it is satisfied. Given a post-condition* Post, *Algorithm 1 always terminates and outputs a mesh M. Let $M_{\mathbb{Z}} = \{val \in \mathbb{Z}^{\mathbb{V}} \mid \exists C \in M \;\; val \in C\}$ be the set of all integer points that appear in the cells of M. We have $M_{\mathbb{Z}} = S_{\mathbb{Z}}$.*

Proof. Each iteration of the while loop at line 5 takes a red cell C and either colors it green or discards it or refines it into two new red cells C_1 and C_2 each with almost half of the integral points in the original cell. Thus, the algorithm terminates. If $val \in M_{\mathbb{Z}}$, then there is a green cell C such that $val \in C$. However, a cell C can be colored green only in lines 9 or 15. In both cases, we have $C \subseteq S_{\mathbb{R}}$ and thus $val \in S_{\mathbb{Z}}$. For the other side, consider an integral valuation $val \in S_{\mathbb{Z}}$. Initially, $val \in C_0$, thus val starts in a red cell. Keep track of the cell that contains val. There will always be such a cell since our refinement does not lose integral points. The cell containing val will never be discarded in line 11 since our check in line 10 is sound. Thus, it has to eventually be colored green and added to the output mesh. □

Subset Checking Procedure. We now present the details of our procedure Is_Subset$(C, \varphi, \mathbb{V})$. Given a cell C and a formula φ which is a Boolean combination of polynomial inequalities, this procedure checks whether every point in C satisfies φ. Equivalently, it checks the implication

$\forall val \in \mathbb{R}^{\mathbb{V}}\ \psi_C \implies \varphi$. As per Theorem 2 we need this check to be sound, but it can be incomplete. In other words, if Is_Subset$(C, \varphi, \mathbb{V})$ returns true, then all points in C must satisfy φ, but the converse is not required. We consider three cases following the grammar in Equation 1:

- **Case 1**: φ has a single linear literal.
  In this case, φ is of the form $f = \alpha_0 + \alpha_1 \cdot v_1 + \alpha_2 \cdot v_2 + \ldots + \alpha_n \cdot v_n \bowtie 0$ where $\bowtie$ is either $>$ or $\geq$ and each α_i is a real constant. We can rewrite the inequalities defining our cell C as follows:

$$\psi_C = \begin{cases} c_1 := v_1 - a_1 \geq 0 \\ c_2 := v_2 - a_2 \geq 0 \\ \vdots \\ c_n := v_n - a_n \geq 0 \\ c_{n+1} := b_1 - v_1 \geq 0 \\ c_{n+2} := b_2 - v_2 \geq 0 \\ \vdots \\ c_{2\cdot n} := b_n - v_n \geq 0 \end{cases} \tag{4}$$

  Our goal is to prove that wherever all linear expressions c_i are non-negative, then f is positive/non-negative. In other words, f is positive/non-negative over the entire cell C. To prove this, we attempt to write f as a linear combination of the c_i's with non-negative coefficients. More formally, our goal is to find $\lambda_0, \ldots, \lambda_{2\cdot n} \geq 0$ such that

$$f = \lambda_0 + \textstyle\sum_{i=1}^{2\cdot n} \lambda_i \cdot c_i. \tag{5}$$

  When aiming to ensure that f is strictly positive over the cell C we additionally require that $\lambda_0 > 0$. Equation 5 leads to a linear programming instance over the λ_i variables. Note that both sides of Equation 5 are linear expressions over $\mathbb{V}$. Thus, they are equal iff they have the same coefficients. We equate the coefficient of each term $\{1, v_1, v_2, \ldots, v_n\}$ in the LHS and RHS of Equation 5. This can be written as

$$\begin{array}{ll} \alpha_0 = \lambda_0 - \sum_{i=1}^{n} \lambda_i \cdot a_i + \sum_{i=1}^{n} \lambda_{n+i} \cdot b_i & \text{constant factor} \\ \alpha_1 = \lambda_1 - \lambda_{n+1} & \text{coefficients of } v_1 \\ \alpha_2 = \lambda_2 - \lambda_{n+2} & \text{coefficients of } v_2 \\ \vdots & \\ \alpha_n = \lambda_n - \lambda_{2\cdot n} & \text{coefficients of } v_n \end{array}$$

  Thus, we obtain a system of linear equations on the variables $\lambda_0, \lambda_1, \cdots, \lambda_{2\cdot n}$ along with the constraints $\lambda_i \geq 0$ for all $1 \leq i \leq 2 \cdot n$ and possibly $\lambda_0 > 0$. This is a linear programming instance and we can use standard LP solvers to check if it has a solution. If a solution exists, we return true. Otherwise, we return false. This case was inspired by the famous Farkas' Lemma from polyhedral geometry [28].

  Example. Consider the literal given by $f := 10 - x - y \geq 0$ and the cell C given by the inequalites $c_1 := x + 5 \geq 0, c_2 := y + 5 \geq 0, c_3 := 5 - x \geq 0, c_4 := 5 - y \geq 0$. We can write f as a non-negative linear combination of the c_i's as follows:

$$10 - x - y = 0 \cdot c_1 + 0 \cdot c_2 + 1 \cdot c_3 + 1 \cdot c_4 = 1 \cdot (5 - x) + 1 \cdot (5 - y)$$

  This imples that f is non-negative over the entire cell C. Therefore, C is a subset of $SAT_{\mathbb{R}}(f)$. See Figure 2 for a visual representation of this example.

- **Case 2**: φ has a single non-linear literal.
  In this case, φ is of the form $f \bowtie 0$ where $\bowtie$ is either $>$ or $\geq$. Moreover, we write the linear inequalities defining our cell C as in the previous case, i.e. Equation 4. We now make a simple observation. At every point in the cell C, every linear expression c_i is non-negative. Thus, so is any product of the c_i's. Suppose we have a fixed positive integer constant d. Let us define

$$\overline{C} := \left\{ \prod_{i=1}^{2\cdot n} c_i^{r_i} \mid \forall i \;\; r_i \in \mathbb{Z} \wedge r_i \geq 0 \wedge \sum_{i=1}^{2\cdot n} r_i \leq d \right\} = \{\overline{c}_1, \ldots, \overline{c}_t\}.$$

  In other words, every $\overline{c}_i$ is a product of at most d different c_i's. Thus, every $\overline{c}_i$ is guaranteed to be non-negative over the entire cell C. We now apply the same technique as in the previous case and attempt to write f as a linear combination of $\overline{c}_i$'s with non-negative coefficients as follows:

$$f = \lambda_0 + \sum_{i=1}^{t} \lambda_i \cdot \overline{c}_i. \tag{6}$$

  Just like the previous case, both sides of Equation 6 are polynomials over $\mathbb{V}$. Thus, for every monomial m appearing in Equation 6, we know that m must have the same coefficients on both the LHS and RHS. Equating the coefficient of m on both sides gives us a linear equation over the λ_i's. Finally, these equations, together with the constraints $\lambda_i \geq 0$, and $\lambda_0 > 0$ in case $\varphi = f > 0$, lead to a linear programming instance. We pass this to an LP solver and return true iff a solution is found. This case was inspired by Handelman's Theorem from real algebraic geometry [35].

  Example. Consider the literal given by $f = 100 - x^2 - y^2 \geq 0$ and the cell C given by the inequalites $c_1 := x + 5 \geq 0, c_2 := y + 5 \geq 0, c_3 := 5 - x \geq 0, c_4 := 5 - y \geq 0$. Now we generate the monomials $\overline{c}_i$ as follows:

$$\overline{C} := \{1, c_1, c_2, c_3, c_4, c_1^2, c_2^2, c_3^2, c_4^2, c_1 \cdot c_2, c_1 \cdot c_3, c_1 \cdot c_4, c_2 \cdot c_3, c_2 \cdot c_4, c_3 \cdot c_4, \cdots\}$$

  We can write f as a non-negative linear combination of $\overline{c}_i$'s as follows:

$$f = 50 \cdot 1 + 1 \cdot c_1 \cdot c_3 + 1 \cdot c_2 \cdot c_4 = 50 + (x+5)(5-x) + (y+5)(5-y).$$

  This implies that f is non-negative over the entire cell C. Therefore, C is a subset of $SAT_{\mathbb{R}}(f)$. See Figure 2 for a visual representation of this example.
- **Case 3**: φ is a Boolean combination of several literals.
  - Step 3.1. Let $\ell_1, \ell_2, \ldots, \ell_k$ be all the literals appearing in φ. We first apply the procedures of the previous cases to every ℓ_i and $\neg\ell_i$. In other words, for every literal ℓ_i, we use the previous cases to see if we can establish that ℓ_i holds at every point in the cell C or that $\neg\ell_i$ holds at every such point.
  - Step 3.2. We write φ in CNF. Thus, φ is a conjunction of clauses who are each a disjunction of a number of literals. If every clause contains a literal that was established in Step 3.1 as holding at every point in C, then φ also holds at every point in C and we return true. Otherwise, we return false.

3.5 Our Synthesis Algorithm

Let us now revisit our original synthesis problem. Given a post-condition Post, our goal is to synthesize the weakest pre-condition Pre for which there exists a program $\mathbf{P}$ such that $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$. As shown in Section 3.2, we have $SAT_{\mathbb{Z}}(\mathsf{Pre}) = \pi\left(SAT_{\mathbb{Z}}(\mathsf{Post})\right)$. Moreover, by Theorem 2, for the mesh M generated by our Algorithm 1 we have $M_{\mathbb{Z}} = S_{\mathbb{Z}} = SAT_{\mathbb{Z}}(\mathsf{Post})$.

In other words, $SAT_{\mathbb{Z}}(\mathsf{Post})$ is precisely the set of integral points that appear in any green cell in the mesh M and $SAT_{\mathbb{Z}}(\mathsf{Pre})$ is just the projection of these points. Fortunately, every cell of our mesh is a hypercube C_i and projecting hypercubes is a simple matter of dropping some constraints. Specifically, for every cell C_i in our mesh we have

$$\psi_{C_i} = \bigwedge_{v_j \in \mathbb{V}} a_{i,j} \leq v_j \leq b_{i,j} \implies \psi_{\pi(C_i)} = \bigwedge_{v_j \in \mathbb{I}} a_{i,j} \leq v_j \leq b_{i,j}.$$

Thus, if $M = \{C_1, C_2, \ldots, C_t\}$ consists of t cells, we compute a new mesh $\pi(M) = \{\pi(C_i) \mid C_i \in M\}$. The integral points in this mesh are precisely $SAT_{\mathbb{Z}}(\mathsf{Pre})$. To compute Pre itself, we just need to translate every cell $\pi(C_i)$ of $\pi(M)$ into a branch of a decision list. Finally, our program $\mathbf{P}$ will have the same structure as our pre-condition Pre, i.e. one branch per cell $\pi(C_i)$, except that in each branch it has to return a value for each output variable in $\mathbb{O}$ that is guaranteed to be in the cell C_i. Specifically, we synthesize the following pre-condition decision list Pre (left) and program $\mathbf{P}$ (right):

```
if ψ_{C_1} then return true
if ψ_{C_2} then return true
      ⋮
if ψ_{C_t} then return true
else return false
```

```
if ψ_{C_1} then return {v_j ↦ a_{1,j} | v_j ∈ 𝕆}
if ψ_{C_2} then return {v_j ↦ a_{2,j} | v_j ∈ 𝕆}
      ⋮
if ψ_{C_t} then return {v_j ↦ a_{t,j} | v_j ∈ 𝕆}
else return fail
```

Theorem 3 (Soundness and Completeness)**.** *Given a set $\mathbb{V} = \mathbb{I} \cup \mathbb{O}$ of bounded integer-valued variables and a specification* Post *which is a Boolean combination of polynomial inequalities over $\mathbb{V}$ (Equation 1), our algorithm always terminates and outputs a pre-condition* Pre *as a simple decision list (Equation 2) and a program $\mathbf{P}$ as a valuation list (Equation 3) such that the Hoare triple* $\{\mathsf{Pre}\}\mathbf{P}\{\mathsf{Post}\}$ *holds. Moreover, it is guaranteed that* Pre *is the weakest pre-condition for which such a program exists.*

Proof. This theorem is a direct consequence of Theorem 2 and the equivalence between synthesis and projection shown in Section 3.2. □

3.6 Further Improvements

In our mesh refinement algorithm, at each step of the while loop in Algorithm 1, we have a cell C in the set of red cells **Red** and a post-condition Post which is a Boolean combination of literals $l_1, l_2, \ldots, l_k$. And we need to perform the following checks: $\textsc{Is\_Subset}(C, \mathsf{Post}, \mathbb{V})$ and $\textsc{Is\_Subset}(C, \neg\mathsf{Post}, \mathbb{V})$. However, these checks can be computationally expensive. We can employ the following two techniques to avoid redundant checks and potentially improve the algorithm's efficiency.

Shadow-aware Mesh Refinement. The idea behind shadow-aware mesh refinement is to maintain a set of cells called **Shadow**, which are the projections of the cells in the set of green cells **Green** onto the input variables $\mathbb{I}$. We then check if the projection $\pi(C)$ of a cell C in **Red** onto the input variables $\mathbb{I}$ is a subset of union of some cells in **Shadow**. If we find that $\pi(C)$ is covered by some subset of cells in **Shadow**, we can safely discard C from **Red**. This is because the cells in **Shadow** already cover the projections of all the integral points in C onto the input variables $\mathbb{I}$. By employing shadow-aware mesh refinement, we can avoid the need for redundant checks and improve the algorithm's efficiency.

- Maintain a set of cells called **Shadow** $= \{\pi(D) \mid D \in$ **Green**$\}$, which are the projections of the cells in the set of green cells **Green** onto the input variables $\mathbb{I}$.

- For cell C in **Red**, compute its projection $\pi(C)$ onto the input variables $\mathbb{I}$.
- Check if the projection $\pi(C)$ is a subset of union of some cells in **Shadow**. If $\pi(C)$ is covered by union of some cells in **Shadow**, discard C from **Red**.

Memoization. The idea behind memoization is to store the results of the checks $\textsc{Is\_Subset}(C, l_i, \mathbb{V})$ and $\textsc{Is\_Subset}(C, \neg l_i, \mathbb{V})$ for each literal l_i and cell C. By doing so, if we find that a cell C is entirely inside or outside $SAT_{\mathbb{R}}(l_i)$, we can conclude that all the refinements of C will also be entirely inside or outside $SAT_{\mathbb{R}}(l_i)$. This eliminates the need to recompute $\textsc{Is\_Subset}(C_1, l_i, \mathbb{V})$ and $\textsc{Is\_Subset}(C_2, l_i, \mathbb{V})$ for the subsequent refinements C_1 and C_2 of C. By employing memoization, we can significantly speed up the process of checking whether a cell C is a subset of $SAT_{\mathbb{R}}(\varphi)$, as it avoids redundant computations during the refinement process.

- Extract the list of literals $l_1, l_2, \ldots, l_k$ from the formula φ.
- Maintain a table T that stores the results of the checks $\textsc{Is\_Subset}(C, l_i, \mathbb{V})$ and $\textsc{Is\_Subset}(C, \neg l_i, \mathbb{V})$ for each literal l_i and cell C.
- If we find that either $\textsc{Is\_Subset}(C, l_i, \mathbb{V})$ or $\textsc{Is\_Subset}(C, \neg l_i, \mathbb{V})$ is true for some i, we memoize this result in table T. This allows us to avoid recomputing $\textsc{Is\_Subset}(C_1, l_i, \mathbb{V})$ and $\textsc{Is\_Subset}(C_2, l_i, \mathbb{V})$ for the further refinements C_1 and C_2 of C in subsequent steps of the algorithm.

4 Experiments

We have implemented our approach in a prototype tool [2] and we have assessed its performance on a suite of benchmarks. Our tool synthesizes both the weakest pre-condition and a program simultaneously from a given post-condition. The pre-conditions synthesized by our tool are decision lists with bound checks on input variables as decision predicates, and the programs are valuation lists with the same bound checks on inputs as decision predicates.

To the best of our knowledge, there are currently no publicly available tools that can simultaneously synthesize pre-conditions and programs over integers, beginning with a post-condition represented as a Boolean combination of polynomials. The work that comes closest to our work is [41]. Their procedure was implemented in a tool called Comfusy [40] which synthesizes the weakest pre-condition and the corresponding program. However, the procedure in [40] seems limited to linear integer arithmetic and some extensions to the non-linear case, which essentially reduces to linear constraints at runtime. We were not successful in our attempts at installing and running Comfusy on our system, and its GitHub page [1] suggests that the tool is currently not maintained. In any case, all the benchmarks that we consider for experiments involve non-linear constraints and fall outside the scope of [40].

Another line of work that addresses a related problem is that of Syntax Guided Synthesis (SyGuS) [5, 7], where the focus is on synthesizing programs starting from (1) a logical specification and (2) (an optional) context-free grammar for the target program that is to be synthesized by the tool, provided as input by the user. Note than unlike our approach, SyGuS tools are not required to generate the weakest pre-condition for the given post-condition, and are allowed to simply bale out with an error message if the weakest pre-condition is not equivalent to `true`.

Since we did not find any other tool that synthesizes both the weakest pre-condition and program for a post-condition that is a Boolean combination of polynomial inequalities over integers, it was not possible to do an apples-to-apples comparison with other tools. For an apples-to-oranges comparison, we considered the state-of-the-art competition-winning SyGuS tool CVC5 [9], and compared the time taken by it to generate a program, *when given the pre-condition synthesized by our tool*, with the time taken by our tool to generate both the program and the weakest pre-condition simultaneously. We ran our experiments on CVC5 with both

the `--sygus` and `--sygus-qe-preproc` flags. The `--sygus` flag is the default one to trigger the SyGuS solver, while the `--sygus-qe-preproc` flag uses quantifier elimination as a preprocessing step before calling the SyGuS solver. We found that, in general, experiments with the latter flag fared much better than in the former case, and therefore we only report the results of these experiments in this paper.

We remark that the programs synthesized by CVC5 are also represented as valuation lists by default, with the notable difference that the conditional expression in the decision predicates often involve non-linear constraints. As a consequence, the programs that are synthesized are quite complex, and not easy to understand. In order to ensure a fair comparison, we considered experiments where we also provided a grammar that would ensure that programs synthesized by CVC5 resemble those generated by our tool, specifically decision lists where the conditionals are of the form of bound checks on input variables. Unfortunately, in this case, CVC5 failed to generate programs within a timeout of 30 minutes. Therefore, we allowed CVC5 to complete the synthesis task without explicitly specifying a grammar (thereby allowing the tool to synthesize any term that it found appropriate for the program), and this turned out to be more efficient.

Benchmarks. Since we didn't find adequate program synthesis benchmarks with polynomial constraints over integers as post-conditions, we created a suite of benchmarks, each of which gives a post-condition represented as a conjunction of polynomial inequalities. In addition, a benchmark includes bound constraints that specify the bounded integral region that we are interested in. More details are in [4, Appendix A].

Experimental Results. If we only provide the post-condition as the input to CVC5, for all our benchmarks, CVC5 fails to synthesize any program. This is because the weakest pre-condition for these benchmarks is not semantically equivalent to `true`. In contrast, our tool synthesizes both the pre-condition and the corresponding program from the post-condition on all our benchmarks.

If we supply the pre-condition generated by our tool to CVC5 as an `assume` statement along with the post-condition, CVC5 is able to synthesize programs adhering to this pre-condition. For this scenario, we report in Table 1 a comparison of the running times of our tool and those of CVC5. Clearly since our tool is solving a more difficult problem than CVC5, in some cases, our tool takes more time than CVC5. Nevertheless, our experiments indicate that our tool, synthesizing both weakest pre-conditoin and program, mostly takes comparable time as CVC5, synthesizing only the program once it is given the weakest pre-condition generated by our tool.

Representation of post-condition. Our tool represents the post-condition after dynamic meshing as a decision tree, and the decision list representing the pre-condition and valuation list representing the program are both derived directly from this tree.

Each node of this tree is labeled a conjunction of bound predicates which checks whether a subset of program variables (inputs and/or outputs) has values lying in a hypercube defined by bounds for each variable. From each node, the outgoing edges indicate the set of predicates that are satisfied. Moreover, the tree is designed such that on any path from root to a leaf, we see predicates on input variables before seeing any predicate on output variables. Then, given a value for input variables we can traverse the tree from the root by choosing from each node the edge based on the evaluation of the predicate. In this way, once we reach a node after evaluating all predicates on input variables, we are left with a sub-tree whose nodes are labeled only by predicates on output variables. Once we reach such a node, we can produce all the solutions (possible values that the output variables can take) by iterating over the leaves of the subtree rooted at this node. The decision list representing the weakest pre-condition is obtained by labeling each such node with "true". Further, the programs that we generate are valuation lists where each such node is labeled by a valuation of outputs that satisfies the

Benchmark	# i/p	# o/p	Running time (in sec)	
			Our tool	CVC5
Example-1	2	1	<1	3
Example-2	1	1	< 1	< 1
Example-3	1	1	< 1	< 1
Example-4	1	1	< 1	4
Example-5	1	1	1	2
Example-6	2	1	12	11
Example-7	2	1	1,440	1,466
Example-8	2	2	828	9
Example-9	2	1	11	12
Example-10	3	1	209	22
Example-11	2	2	245	187
Example-12	1	1	2	26
Example-13	2	1	34	126
Example-14	3	1	55	5
Example-15	2	2	54	t/o
Example-16	1	1	< 1	< 1

Table 1: Experimental results obtained by running CVC5 and our prototype tool on various benchmark examples. The experiments were run on a HP-Desktop-Pro-G1-MT with 3.2 GHz Intel Core i7-8700 processor and 32 GB of memory. The timeout is 30 minutes. For our tool, running time refers to the time taken to generate both the pre-conditions and the program. In the case of CVC5, running time refers to the time taken to synthesize the program when the pre-condition is given.

bound predicates along one of the paths leading down from this node to a leaf.

Note that using the above representation, one can easily generate the set of all integral output values that serve to satisfy the specification for each value of the inputs. This is particularly useful in the setting where the user needs to select (based on some metric that is not known apriori) a point from the set of all points that satisfy the specification.

5 Conclusion

In this paper, we proposed an exact procedure that simultaneously synthesizes pre-conditions and programs over integers, starting from a post-condition that is a Boolean combination of polynomials. Since the general variant of the problem is undecidable, we consider the variant of the problem where we restrict the region of interest to a bounded domain for the integral values. Our procedure synthesizes the programs and pre-conditions as decision lists, which are amenable for further static analysis. We also provide experimental evaluation of our approach and provide comparison of the performance w.r.t. SyGuS tools, which are the closest to our approach.

Acknowledgments and Notes

The research was partially supported by the Asian Universities Alliance Scholars Award Program (AUASAP), which financed a visit by S. Akshay to HKUST and another visit by A.K. Goharshady to IIT Bombay, as well as the Hong Kong Research Grants Council (RGC) ECS Project Number 26208122. The authors are grateful to the Schloss Dagstuhl – Leibniz Center for Informatics. This collaboration started at the Dagstuhl Seminar 23241: "Scalable Analysis of Probabilistic Models and Programs". Authors are ordered alphabetically by last name.

References

[1] Github page of Comfusy. https://github.com/epfl-lara/comfusy.

[2] Github page of our tool. https://github.com/tejavaranasi2/LPAR.

[3] S. Akshay, Supratik Chakraborty, Shubham Goel, Sumith Kulal, and Shetal Shah. Boolean functional synthesis: hardness and practical algorithms. *Formal Methods Syst. Des.*, 57(1):53–86, 2021.

[4] S Akshay, Supratik Chakraborty, Amir Kafshdar Goharshady, R Govind, Harshit Jitendra Motwani, and Sai Teja Varanasi. Automated synthesis of decision lists for polynomial specifications over integers. https://hal.science/hal-04568906, 2024.

[5] 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, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.

[7] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61(12):84–93, 2018.

[8] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. Polynomial reachability witnesses via Stellensätze. In *PLDI*, pages 772–787, 2021.

[9] 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 *TACAS*, pages 415–442, 2022.

[10] Zhuo Cai, Soroush Farokhnia, Amir Kafshdar Goharshady, and S. Hitarth. Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts. *Proc. ACM Program. Lang.*, 7(OOPSLA2):882–911, 2023.

[11] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. Flashfill++: Scaling programming by example by cutting to the chase. *Proc. ACM Program. Lang.*, 7(POPL):952–981, 2023.

[12] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Decision diagrams for linear arithmetic. In *FMCAD*, pages 53–60, 2009.

[13] Supratik Chakraborty and S. Akshay. On synthesizing computable Skolem functions for first order logic. In *MFCS*, pages 30:1–30:15, 2022.

[14] Supratik Chakraborty, Dror Fried, Lucas M. Tabajara, and Moshe Y. Vardi. Functional synthesis via input-output separation. *Formal Methods Syst. Des.*, 60(2):228–258, 2022.

[15] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through Positivstellensatz's. In *CAV*, pages 3–22, 2016.

[16] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Non-polynomial worst-case analysis of recursive programs. In *CAV*, pages 41–63, 2017.

[17] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Non-polynomial worst-case analysis of recursive programs. *ACM Trans. Program. Lang. Syst.*, 41(4):20:1–20:52, 2019.

[18] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*, pages 672–687, 2020.

[19] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Nastaran Okati. Computational approaches for stochastic shortest path on succinct mdps. In *IJCAI*, pages 4700–4707, 2018.

[20] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Ehsan Kafshdar Goharshady, Mehrdad Karrabi, and Djordje Zikelic. Sound and complete witnesses for template-based verification of LTL properties on polynomial programs. *CoRR*, abs/2403.05386, 2024.

[21] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Djordje Zikelic. Quantitative bounds on resource usage of probabilistic programs. In *OOPSLA*, 2024.

[22] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. Sound and complete certificates for quantitative termination analysis of probabilistic programs. In *CAV*, pages 55–78, 2022.

[23] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In H. Brakhage, editor, *Automata Theory and Formal Languages*. Springer, 1975.

[24] George E. Collins. Quantifier elimination by cylindrical algebraic decomposition - twenty years of progress. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 8–23, 1998.

[25] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[26] Martin Davis, Yuri Matijasevic, and Julia Robinson. Hilbert's tenth problem. diophantine equations: positive aspects of a negative solution. In *Proceedings of symposia in pure mathematics*, volume 28, pages 323–378, 1976.

[27] Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. SAT-based strategy extraction in reachability games. In *AAAI*, pages 3738–3745, 2015.

[28] Julius Farkas. Theory of simple inequalities. *Journal for pure and applied mathematics (Crelles Journal)*, 1902(124):1–27, 1902.

[29] Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Special issue on syntax-guided synthesis preface. *Formal Methods Syst. Des.*, 58(3):469–470, 2021.

[30] Amir Kafshdar Goharshady. *Parameterized and Algebro-geometric Advances in Static Program Analysis*. PhD thesis, Institute of Science and Technology Austria, Klosterneuburg, Austria, 2020.

[31] Amir Kafshdar Goharshady, S. Hitarth, Fatemeh Mohammadi, and Harshit J. Motwani. Algebro-geometric algorithms for template-based synthesis of polynomial programs. In *OOPSLA*, pages 727–756, 2023.

[32] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Program synthesis as dependency quantified formula modulo theory. In *IJCAI*, pages 1894–1900, 2021.

[33] Sumit Gulwani and Prateek Jain. Programming by examples: PL meets ML. In *APLAS*, pages 3–20, 2017.

[34] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*. August 2017.

[35] David Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–62, 1988.

[36] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174, 2020.

[37] Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):129:1–129:29, 2019.

[38] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[39] Mikolás Janota. Towards generalization in QBF solving via machine learning. In *AAAI*, pages 6607–6614, 2018.

[40] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, pages 430–433, 2010.

[41] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[42] Hongming Liu, Hongfei Fu, Zhiyong Yu, Jiaxin Song, and Guoqiang Li. Scalable linear invariant generation with Farkas' lemma. *Proc. ACM Program. Lang.*, 6(OOPSLA2):204–232, 2022.

[43] Kuldeep S. Meel. Counting, sampling, and synthesis: The quest for scalability. In *IJCAI*, pages 5816–5820, 2022.

[44] Harshit Jitendra Motwani. *Algebro-Geometric Algorithms for Program Synthesis, Tensor Networks, and Conditional Independence Models.* PhD thesis, Ghent University, Ghent, Belgium, 2023.

[45] Markus N. Rabe. Incremental determinization for quantifier elimination and functional synthesis. In *CAV*, pages 84–94, 2019.

[46] Ronald L Rivest. Learning decision lists. *Machine learning*, 2:229–246, 1987.

[47] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.

[48] Mark Santolucito, Drew Goldman, Allyson Weseley, and Ruzica Piskac. Programming by example: Efficient, but not "Helpful". In *PLATEAU@SPLASH*, volume 67 of *OASIcs*, pages 3:1–3:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[49] A. Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 60(2):365–374, 1954.

[50] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.

[51] Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Automated tail bound analysis for probabilistic recurrence relations. In *CAV*, pages 16–39, 2023.

[52] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry: Prepared for Publication with the Assistance of J.C.C. McKinsey.* RAND Corporation, Santa Monica, CA, 1951.

[53] Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Quantitative analysis of assertion violations in probabilistic programs. In *PLDI*, pages 1171–1186, 2021.

[54] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. Cost analysis of nondeterministic probabilistic programs. In *PLDI*, pages 204–220, 2019.