Program Synthesis in Saturation

Petra Hozzová, Laura Kovács, Chase Norman and Andrei Voronkov

May 21, 2023

Program Synthesis in Saturation

Petra Hozzová[1], Laura Kovács[1], Chase Norman[2], and Andrei Voronkov[3,4]

[1] TU Wien
[2] UC Berkeley
[3] University of Manchester
[4] EasyChair

Abstract. We present an automated reasoning framework for synthesizing recursion-free programs using saturation-based theorem proving. Given a functional specification encoded as a first-order logical formula, we use a first-order theorem prover to both establish validity of this formula and discover program fragments satisfying the specification. As a result, when deriving a proof of program correctness, we also synthesize a program that is correct with respect to the given specification. We describe properties of the calculus that a saturation-based prover capable of synthesis should employ, and extend the superposition calculus in a corresponding way. We implemented our work in the first-order prover VAMPIRE, extending the successful applicability of first-order proving to program synthesis.

Keywords: Program Synthesis · Saturation · Superposition · Theorem Proving.

1 Introduction

Program synthesis constructs code from a given specification. In this work we focus on synthesis using functional specifications summarized by valid first-order formulas [13,1], ensuring that our programs are provably correct. While being a powerful alternative to formal verification [19], synthesis faces intrinsic computational challenges. One of these challenges is posed to the reasoning backend used for handling program specifications, as the latter typically include first-order quantifier alternations and interpreted theory symbols. As such, efficient reasoning with both theories and quantifiers is imperative for any effort towards program synthesis.

In this paper we address this demand for recursion-free programs. We advocate the use of first-order theorem proving for extracting code from correctness proofs of functional specifications given as first-order formulas $\forall \overline{x}. \exists y. F[\overline{x}, y]$. These formulas state that "for all (program) inputs $\overline{x}$ there exists an output y such that the input-output relation (program computation) $F[\overline{x}, y]$ is valid". Given such a specification, we synthesize a recursion-free program while also deriving a proof certifying that the program satisfies the specification.

The programs we synthesize are built using first-order theory terms extended with `if−then−else` constructors. To ensure that our programs yield computational models, i.e., that they can be evaluated for given values of input variables $\overline{x}$, we restrict the programs we synthesize to only contain *computable* symbols.

Our approach in a nutshell. In order to synthesize a recursion-free program, we prove its functional specification using saturation-based theorem proving [14,10]. We extend saturation-based proof search with answer literals [5], allowing us to track substitutions into the output variable y of the specification. These substitutions correspond to the sought program fragments and are conditioned on clauses they are associated with in the proof. When we derive a clause corresponding to a program branch $\texttt{if}\ C\ \texttt{then}\ r$, where C is a condition and r a term and both C, r are computable, we store it and continue proof search assuming that $\neg C$ holds; we refer to such conditions C as (program) branch conditions. The saturation process for both proof search and code construction terminates when the conjunction of negations of the collected branch conditions becomes unsatisfiable. Then we synthesize the final program satisfying the given (and proved) specification by assembling the recorded program branches (see e.g. Examples 1- 3).

The main challenges of making our approach effective come with (i) integrating the construction of the programs with `if − then − else` into the proof search, turning thus proof search into *program search/synthesis*, and (ii) guiding program synthesis to only computable branch conditions and programs.

Contributions. We bring the next contributions solving the above challenges:[5]

- We formalize the semantics for clauses with answer literals and introduce a *saturation-based algorithm for program synthesis* based on this semantics. We prove that, given a sound inference system, our saturation algorithm derives correct and computable programs (Section 4).
- We define properties of a sound inference calculus in order to make the calculus suitable for our saturation-based algorithm for program synthesis. We accordingly extend the superposition calculus and define a class of substitutions to be used within the extended calculus; we refer to these substitutions as *computable unifiers* (Section 5).
- We extend a first-order unification algorithm to find computable unifiers (Section 6) to be further used in saturation-based program synthesis.
- We implement our work in the Vampire prover [10] and evaluate our synthesis approach on a number of examples, complementing other techniques in the area (Section 7). For example, our results demonstrate the applicability of our work on synthesizing programs for specifications that cannot be even encoded in the SyGuS syntax [15].

[5] proofs of our results are given in Appendix A

2 Preliminaries

We assume familiarity with standard multi-sorted first-order logic with equality. We denote variables by x, y, terms by s, t, atoms by A, literals by L, clauses by C, D, formulas by F, G, all possibly with indices. Further, we write σ for Skolem constants. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. Formulas and clauses with free variables are considered implicitly universally quantified (i.e. we consider closed formulas). By $\simeq$ we denote the equality predicate and write $t \not\simeq s$ as a shorthand for $\neg t \simeq s$. We use a distinguished *integer sort*, denoted by $\mathbb{Z}$. When we use standard integer predicates $<$, $\leq$, $>$, $\geq$, functions $+, -, \ldots$ and constants $0, 1, \ldots$, we assume that they denote the corresponding interpreted integer predicates and functions with their standard interpretations. Additionally, we include a conditional term constructor $\texttt{if}-\texttt{then}-\texttt{else}$ in the language, as follows: given a formula F and terms s, t of the same sort, we write $\texttt{if}\ F\ \texttt{then}\ s\ \texttt{else}\ t$ to denote the term s if F is valid and t otherwise.

An *expression* is a term, literal, clause or formula. We write $E[t]$ to denote that the expression E contains the term t. For simplicity, $E[s]$ denotes the expression E where all occurrences of t are replaced by the term s. A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two expressions E and E' if $E\theta = E'\theta$, and is a *most general unifier* (*mgu*) if for every unifier η of E and E', there exists substitution μ such that $\eta = \theta\mu$. We denote the mgu of E and E' with $\texttt{mgu}(E, E')$. We write $F_1, \ldots, F_n \vdash G_1, \ldots, G_m$ to denote that $F_1 \wedge \ldots \wedge F_n \rightarrow G_1 \vee \ldots \vee G_m$ is valid, and extend the notation also to validity modulo a theory T. Symbols occurring in a theory T are *interpreted* and all other symbols are *uninterpreted*.

2.1 Computable Symbols and Programs

We distinguish between *computable* and *uncomputable* symbols in the signature. The set of computable symbols is given as part of the specification. Intuitively, a symbol is computable if it can be evaluated and hence is allowed to occur in a synthesized program. A term or a literal is *computable* if all symbols it contains are computable. A symbol, term or literal is *uncomputable* if it is not computable.

A *functional specification*, or simply just a *specification*, is a formula

$$\forall \overline{x}. \exists y. F[\overline{x}, y]. \tag{1}$$

The variables $\overline{x}$ of a specification (1) are called *input variables.* Note that while we use specifications with a single variable y, our work can analogously be used with a tuple of variables $\overline{y}$ in (1).

Let $\overline{\sigma}$ denote a tuple of Skolem constants. Consider a computable term $r[\overline{\sigma}]$ such that the instance $F[\overline{\sigma}, r[\overline{\sigma}]]$ of (1) holds. Since $\overline{\sigma}$ are fresh Skolem constants, the formula $\forall \overline{x}. F[\overline{x}, r[\overline{x}]]$ also holds; we call such $r[\overline{x}]$ a *program* for (1) and say that the program $r[\overline{x}]$ *computes a witness* of (1).

Superposition:

$$\frac{\underline{s \simeq t} \vee C \quad \underline{L[s']} \vee C'}{(L[t] \vee C \vee C')\theta} \qquad \frac{\underline{s \simeq t} \vee C \quad \underline{u[s'] \not\simeq u'} \vee C'}{(u[t] \not\simeq u' \vee C \vee C')\theta} \qquad \frac{\underline{s \simeq t} \vee C \quad \underline{u[s'] \simeq u'} \vee C'}{(u[t] \simeq u' \vee C \vee C')\theta}$$

where $\theta := \mathtt{mgu}(s, s')$; $t\theta \not\succeq s\theta$; (first rule only) $L[s']$ is not an equality literal; and (second and third rules only) $u'\theta \not\succeq u[s']\theta$.

Binary resolution:

$$\frac{\underline{A} \vee C \quad \neg\underline{A'} \vee C'}{(C \vee C')\theta}$$

where $\theta := \mathtt{mgu}(A, A')$.

Factoring:

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta}$$

where $\theta := \mathtt{mgu}(A, A')$.

Equality resolution:

$$\frac{\underline{s \not\simeq t} \vee C}{C\theta}$$

where $\theta := \mathtt{mgu}(s, t)$.

Equality factoring:

$$\frac{\underline{s \simeq t} \vee \underline{s' \simeq t'} \vee C}{(s \simeq t \vee t \not\simeq t' \vee C)\theta}$$

where $\theta := \mathtt{mgu}(s, s')$; $t\theta \not\succeq s\theta$; and $t'\theta \not\succeq t\theta$.

Fig. 1. The superposition calculus $\mathbb{S}\mathrm{up}$.

Further, if $\forall \overline{x}.(F_1 \wedge \ldots \wedge F_n \rightarrow F[\overline{x}, r[\overline{x}]])$ holds for computable formulas $F_1, \ldots, F_n$, we write $\langle r[\overline{x}], \bigwedge_{i=1}^{n} F_i \rangle$ to refer to a *program with conditions* $F_1, \ldots, F_n$ for (1). In the sequel, we refer to (parts of) programs with conditions also as *conditional branches*. In Section 4 we show how to build programs for (1) by composing programs with conditions for (1) (see Corollary 3).

2.2 Saturation and Superposition

Saturation-based proof search implements *proving by refutation* [10]: to prove validity of F, a saturation algorithm establishes unsatisfiability of $\neg F$. First-order theorem provers work with clauses, rather than with arbitrary formulas. To prove a formula F, first-order provers negate F which is further skolemized and converted to clausal normal form (CNF). The CNF of $\neg F$ is denoted by $\mathtt{cnf}(\neg F)$ and represents a set S of initial clauses. First-order provers then *saturate* S by computing logical consequences of S with respect to a sound inference system $\mathcal{I}$. The saturated set of S is called the *closure* of S and the process of computing the closure of S is called *saturation*. If the closure of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable, and hence the formula F is valid.

We may extend the set S of initial clauses with additional clauses $C_1, \ldots, C_n$. If C is derived by saturating this extended set, we say C is derived from S *under additional assumptions* $C_1, \ldots, C_n$.

The *superposition calculus*, denoted as $\mathbb{S}\mathrm{up}$ and given in Figure 1, is the most common inference system used by saturation-based provers for first-order logic with equality [14]. The $\mathbb{S}\mathrm{up}$ calculus is parametrized by a *simplification ordering* $\succ$ on terms and a *selection function*, which selects in each non-empty clause a non-empty subset of literals (possibly also positive literals). We denote selected literals by underlining them. An inference rule can be applied on the given premise(s) if the literals that are underlined in the rule are also selected in the premise(s). For a certain class of selection functions, the superposition calculus $\mathbb{S}\text{up}$ is *sound* (if $\square$ is derived from F, then F is unsatisfiable) and *refutationally complete* (if F is unsatisfiable, then $\square$ can be derived from it).

2.3 Answer Literals

Answer literals [5] provide a question answering technique for tracking substitutions into given variables throughout the proof. Suppose we want to find a witness for the validity of the formula

$$\exists y.F[y]. \tag{2}$$

Within saturation-based proving, we first derive the skolemized negation of (2) and add an *answer literal* using a fresh predicate $\texttt{ans}$ with argument y, yielding

$$\forall y.(\neg F[y] \vee \texttt{ans}(y)). \tag{3}$$

We then saturate the CNF of (3), while ensuring that answer literals are not selected for performing inferences. If the clause $\texttt{ans}(t_1) \vee \ldots \vee \texttt{ans}(t_m)$ is derived during saturation, note that this clause contains only answer literals in addition to the empty clause; hence, in this case we proved unsatisfiability of $\forall y.\neg F[y]$, implying validity of (2). Moreover, $t_1, \ldots, t_m$ provides a *disjuntive answer*, i.e. witness, for the validity of (2); that is, $F[t_1] \vee \ldots \vee F[t_m]$ holds [11]. In particular, if we derive the clause $\texttt{ans}(t)$ during saturation, we found a *definite answer* t for (2), namely $F[t]$ is valid.

Answer literals with $\texttt{if}-\texttt{then}-\texttt{else}$. The derivation of disjunctive answers can be avoided by modifying the inference rules to only derive clauses containing at most one answer literal. One such modification is given within the A(R)-calculus for binary resolution [21], where R is a so-called strongly liftable term restriction. The A(R)-calculus replaces the binary resolution rule when both premises contain an answer literal by the following A-resolution rule:

$$\frac{A \vee C \vee \texttt{ans}(r) \quad \neg A' \vee C' \vee \texttt{ans}(r')}{(C \vee C' \vee \texttt{ans}(\texttt{if } A \texttt{ then } r' \texttt{ else } r))\theta} \text{ (A-resolution)},$$

where $\theta := \texttt{mgu}(A, A')$ and the restriction $R(\texttt{if } A \texttt{ then } r' \texttt{ else } r)$ holds.

In our work we go beyond the A-resolution rule and modify both the superposition calculus and the saturation algorithm to reason not only about answer literals but also about their use of $\texttt{if}-\texttt{then}-\texttt{else}$ terms (see Sections 4–5).

3 Illustrative Example

Let us illustrate our approach to program synthesis. We use answer literals in saturation to construct programs with conditions while proving specifications (1). By adding an answer literal to the skolemized negation of (1), we obtain

$$\forall y.(\neg F[\overline{\sigma}, y] \vee \texttt{ans}(y)), \tag{4}$$

$\forall x.\, i(x)*x \simeq e$ (A1) $\quad \forall x.\, e*x \simeq x$ (A2) $\quad \forall x, y, z.\, x*(y*z) \simeq (x*y)*z$ (A3)

Fig. 2. Axioms defining a group. Uninterpreted function symbols $i(\cdot), e, *$ represent the inverse, the identity element, and the group operation, respectively.

where $\overline{\sigma}$ are the skolemized input variables x. When we derive a unit clause $\texttt{ans}(r[\overline{\sigma}])$ during saturation, where $r[\overline{\sigma}]$ is a computable term, we construct a program for (1) from the definite answer $r[\overline{\sigma}]$ by replacing $\overline{\sigma}$ with the input variables $\overline{x}$, obtaining the program $r[\overline{x}]$. Hence, deriving computable definite answers by saturation allows us to synthesize programs for specifications.

Example 1. Consider the group theory axioms (A1)–(A3) of Figure 2. We are interested in synthesizing a program for the following specification:

$$\forall x.\exists y.\ x * y \simeq e \tag{5}$$

In this example we assume that all symbols are computable. To synthesize a program for (5), we add an answer literal to the skolemized negation of (5) and convert the resulting formula to CNF (preprocessing). We consider the set S of clauses containing the obtained CNF and the axioms (A1)-(A3). We saturate S using $\mathbb{S}\text{up}$ and obtain the following derivation:[6]

1. $\sigma * y \not\simeq e \vee \texttt{ans}(y)$ [preprocessed specification]
2. $i(x) * (x * y) \simeq e * y$ [Sup A1, A3]
3. $i(x) * (x * y) \simeq y$ [Sup A2, 2.]
4. $x * y \simeq i(i(x)) * y$ [Sup 3., 3.]
5. $e \simeq x * i(x)$ [Sup 4., A1]
6. $\texttt{ans}(i(\sigma))$ [BR 5., 1.]

Using the above derivation, we construct a program for the functional specification (5) as follows: we replace σ in the definite answer $i(\sigma)$ by x, yielding the program $i(x)$. Note that for each input x, our synthesized program computes the inverse $i(x)$ of x as an output. In other words, our synthesized program for (5) ensures that each group element x has a right inverse $i(x)$.

While Example 1 yields a definite answer within saturation-based proof search, our work supports the synthesis of more complex recursion-free programs (see Examples 2–3) by composing program fragments derived in the program search (Section 4) as well as by using answer literals with $\texttt{if}-\texttt{then}-\texttt{else}$ to effectively handle disjunctive answers (Section 5).

[6] For each formula in the derivation, we also list how the formula has been derived. For example, formula 5 is the result of superposition (Sup) with formula 4 and axiom A1, whereas binary resolution (BR) has been used to derive formula 6 from 5 and 1.

4 Program Synthesis with Answer Literals

We now introduce our approach to saturation-based program synthesis using answer literals (Algorithm 1). We focus on recursion-free program synthesis and present our work in a more general setting. Namely, we consider functional specifications whose validity may depend on additional assumptions (e.g. additional program requirements) $A_1, \ldots, A_n$, where each A_i is a closed formula:

$$A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x}. \exists y. F[\overline{x}, y] \tag{6}$$

Note that specification (1) is a special case of (6). However, since $A_1, \ldots, A_n$ are closed formulas, (6) is equivalent to $\forall \overline{x}. \exists y. (A_1 \wedge \ldots \wedge A_n \rightarrow F[\overline{x}, y])$, which is a special case of (1).

Given a functional specification (6), we use answer literals to synthesize programs with conditions (Section 4.1) and extend saturation-based proof search to reason about answer literals (Section 4.2). For doing so, we add the answer literal $\mathtt{ans}(y)$ to the skolemized negation of (6) and obtain

$$A_1 \wedge \ldots \wedge A_n \wedge \forall y. (\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y)). \tag{7}$$

We saturate the CNF of (7), while ensuring that answer literals are not selected within the inference rules used in saturation. We guide saturation-based proof search to derive clauses $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$, where $C[\overline{\sigma}]$ and $r[\overline{\sigma}]$ are computable.

4.1 From Answer Literals to Programs

Our next result ensures that, if we derive the clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$, the term $r[\overline{\sigma}]$ is a definite answer under the assumption $\neg C[\overline{\sigma}]$ (Theorem 1). We note that we do not terminate saturation-based program synthesis once a clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived. We rather record the program $r[\overline{x}]$ with condition $\neg C[\overline{x}]$ (and possibly also other conditions), replace clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ by $C[\overline{\sigma}]$, and continue saturation (Corollary 2). As a result, upon establishing validity of (6), we synthesized a program for (6) (Corollary 3).

Theorem 1 [Semantics of Clauses with Answer Literals] *Let C be a clause not containing an answer literal. Assume that, using a saturation algorithm based on a sound inference system $\mathcal{I}$, the clause $C \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived from the set of clauses consisting of initial assumptions $A_1, \ldots, A_n$, the clausified formula $\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y))$ and additional assumptions $C_1, \ldots, C_m$. Then,*

$$A_1, \ldots, A_n, C_1, \ldots, C_m \vdash C, F[\overline{\sigma}, r[\overline{\sigma}]].$$

That is, under the assumptions $C_1, \ldots, C_m, \neg C$, the computable term $r[\overline{\sigma}]$ provides a definite answer to (6).

We further use Theorem 1 to synthesize programs with conditions for (6).

Corollary 2 [Programs with Conditions] *Let $r[\overline{\sigma}]$ be a computable term and $C[\overline{\sigma}]$ a ground computable clause not containing an answer literal. Assume that clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived from the set of initial clauses $A_1, \ldots, A_n$, the clausified formula $\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y))$ and additional ground computable assumptions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$, by using saturation based on a sound inference system $\mathcal{I}$. Then,*

$$\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$$

is a program with conditions for (6)*.*

Note that a program with conditions $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$ corresponds to a conditional (program) branch `if` $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]$ `then` $r[\overline{x}]$: only if the condition $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]$ is valid, then $r[\overline{x}]$ is computed for (6).

We use programs with conditions $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}] \rangle$ to finally synthesize a program for (6). To this end, we use Corollary 2 to derive programs with conditions, and once their conditions cover all possible cases given the initial assumptions $A_1, \ldots, A_n$, we compose them into a program for (6).

Corollary 3 [From Programs with Conditions to Programs for (6)**]** *Let $P_1[\overline{x}], \ldots, P_k[\overline{x}]$, where $P_i[\overline{x}] = \langle r_i[\overline{x}], \bigwedge_{j=1}^{i-1} C_j[\overline{x}] \wedge \neg C_i[\overline{x}] \rangle$, be programs with conditions for* (6)*, such that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable. Then $P[\overline{x}]$, given by*

$$\begin{aligned} P[\overline{x}] := \ & \mathtt{if}\ \neg C_1[\overline{x}]\ \mathtt{then}\ r_1[\overline{x}] \\ & \mathtt{else\ if}\ \neg C_2[\overline{x}]\ \mathtt{then}\ r_2[\overline{x}] \\ & \quad \ldots \\ & \mathtt{else\ if}\ \neg C_{k-1}[\overline{x}]\ \mathtt{then}\ r_{k-1}[\overline{x}] \\ & \mathtt{else}\ r_k[\overline{x}], \end{aligned} \tag{8}$$

is a program for (6)*.*

Note that since the conditional branches of (8) cover all possible cases to be considered over $\overline{x}$, we do not need the condition `if` $\neg C_k$. In particular, if $k = 1$, i.e. $\bigwedge_{i=1}^{n} A_i \wedge C_1[\overline{x}]$ is unsatisfiable, then the synthesized program for (6) is $r_1[\overline{x}]$.

4.2 Saturation-Based Program Synthesis

Our program synthesis results from Theorem 1, Corollary 2 and Corollary 3 rely upon a saturation algorithm using a sound (but not necessarily complete) inference system $\mathcal{I}$. In this section, we present our modifications to extend state-of-the-art saturation algorithms with answer literal reasoning, allowing to derive clauses $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$, where both $C[\overline{\sigma}]$ and $r[\overline{\sigma}]$ are computable. In Sections 5–6 we then describe modifications of the inference system $\mathcal{I}$ to implement rules over clauses with answer literals.

Algorithm 1 Saturation Loop for Recursion-Free Program Synthesis

1 initial set of clauses $S := \{\texttt{cnf}(A_1 \wedge \ldots \wedge A_n \wedge \forall y.(\neg F[\overline{\sigma}, y] \vee \texttt{ans}(y)))\}$
2 initial sets of additional assumptions $\mathcal{C} := \emptyset$ and programs $\mathcal{P} := \emptyset$
3 **repeat**
4   Select clause $G \in S$
5   Derive consequences $C_1, \ldots, C_n$ of G and formulas from S using rules of $\mathcal{I}$
6   **for each** C_i **do**
7   **if** $C_i = (C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}]))$ and $C[\overline{\sigma}]$ is ground and computable **then**
8             $\mathcal{P} := \mathcal{P} \cup \{\langle r[\overline{x}], \bigwedge_{C' \in \mathcal{C}} C' \wedge \neg C[\overline{x}] \rangle\}$ /* Corollary 2 */
9             $\mathcal{C} := \mathcal{C} \cup \{C[\overline{x}]\}$
10             $C_i := C[\overline{\sigma}]$
11     $S := S \cup \{C_1, \ldots, C_n\}$
12   **if** $\Box \in S$ **then**
13   **return** program (8) for specification (6), derived from $\mathcal{P}$ /* Corollary 3 */

Our saturation algorithm is given in Algorithm 1. In a nutshell, we use Corollary 2 to construct programs from clauses $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$ and replace clauses $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$ by $C[\overline{\sigma}]$ (lines 7–10 of Algorithm 1). The newly added computable assumptions $C[\overline{\sigma}]$ are used to guide saturation towards deriving programs with conditions where the conditions contain $C[\overline{x}]$; these programs with conditions are used for synthesizing programs for (6), as given in Corollary 3.

Compared to a standard saturation algorithm used in first-order theorem proving (e.g. lines 4–5 of Algorithm 1), Algorithm 1 implements additional steps for processing newly derived clauses $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$ with answer literals (lines 6-10). As a result, Algorithm 1 establishes not only the validity of the specification (6) but also synthesizes a program (lines 12-13). Throughout the algorithm, we maintain a set $\mathcal{P}$ of programs with conditions derived so far and a set $\mathcal{C}$ of additional assumptions. For each new clause C_i, we check if it is in the form $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$ where $C[\overline{\sigma}]$ is ground and computable (line 7). If yes, we construct a program with conditions $\langle r[\overline{x}], \bigwedge_{C' \in \mathcal{C}} C' \wedge \neg C[\overline{x}] \rangle$, extend $\mathcal{C}$ with the additional assumption $C[\overline{x}]$, and replace C_i by $C[\overline{\sigma}]$ (lines 8-10). Then, when we derive the empty clause, we construct the final program as follows. We first collect all clauses that participated in the derivation of $\Box$. We use this clause collection to filter the programs in $\mathcal{P}$ – we only keep a program originating from a clause $C[\overline{\sigma}] \vee \texttt{ans}(r[\overline{\sigma}])$ if the condition $C[\overline{\sigma}]$ was used in the proof, obtaining programs $P_1, \ldots, P_k$. From $P_1, \ldots, P_k$ we then synthesize the final program P using the construction (8) from Corollary 3.

Remark 1. Compared to [21] where potentially large programs (with conditions) are tracked in answer literals, Algorithm 1 removes answer literals from clauses and constructs the final program only after saturation found a refutation of the negated (6). Our approach has two advantages: first, we do not have to keep track of potentially many large terms using `if−then−else`, which might slow down saturation-based program synthesis. Second, our work can naturally be integrated with clause splitting techniques within saturation (see Section 7).

5 Superposition with Answer Literals

We note that our saturation-based program synthesis approach is not restricted to a specific calculus. Algorithm 1 can thus be used with *any sound* set of inference rules, including theory-specific inference rules, e.g. [9], as long as the rules allow derivation of clauses in the form $C \vee \mathtt{ans}(r)$, where C, r are computable and C is ground. I.e., the rules should only derive clauses with at most one answer literal, and should not introduce uncomputable symbols into answer literals.

In this section we present changes tailored to the superposition calculus $\mathbb{S}\mathrm{up}$, yet, without changing the underlying saturation process of Algorithm 1. We first introduce the notion of an abstract unifier [16] and define a computable unifier – a mechanism for dealing with the uncomputable symbols in the reasoning instead of introducing them into the programs. The use of such a unifier in any sound calculus is explained, with particular focus on the $\mathbb{S}\mathrm{up}$ calculus.

Definition 1 (Abstract unifier [16]). *An* abstract unifier *of two expressions* E_1, E_2 *is a pair* (θ, D) *such that:*

1. θ *is a substitution and* D *is a (possibly empty) disjunction of disequalities,*
2. $(D \vee E_1 \simeq E_2)\theta$ *is valid in the underlying theory.*

Intuitively speaking, an abstract unifier combines disequality constraints D with a substitution θ such that the substitution is a unifier of E_1, E_2 if the constraints D are not satisfied.

Definition 2 (Computable unifier). *A* computable unifier *of two expressions* E_1, E_2 *with respect to an expression* E_3 *is an abstract unifier* (θ, D) *of* E_1, E_2 *such that the expression* $E_3\theta$ *is computable.*

For example, let f be computable and g uncomputable. Then $(\{y \mapsto f(z)\}, z \not\simeq g(x))$ is a computable unifier of the terms $f(g(x)), y$ with respect to $f(y)$. Further, $(\{y \mapsto f(g(x))\}, \emptyset)$ is an abstract unifier of the same terms, but not a computable unifier with respect to $f(y)$.

Ensuring computability of answer literal arguments. We modify the rules of a sound inference system $\mathcal{I}$ to use computable unifiers with respect to the answer literal argument instead of unifiers. Since a computable unifier may entail disequality constraints D, we add D to the conclusions of the inference rules. That is, for an inference rule of $\mathcal{I}$ as below

$$\frac{C_1 \quad \cdots \quad C_n}{C\theta}\ , \tag{9}$$

where θ is a substitution such that $E\theta \simeq E'\theta$ holds for some expressions E, E', we extend $\mathcal{I}$ with the following n inference rules with computable unifiers:

$$\frac{C_1 \vee \mathtt{ans}(r) \quad C_2 \quad \cdots \quad C_n}{(\underline{D} \vee C \vee \mathtt{ans}(r))\theta'} \quad \cdots \quad \frac{C_1 \quad C_2 \quad \cdots \quad C_n \vee \mathtt{ans}(r)}{(\underline{D} \vee C \vee \mathtt{ans}(r))\theta'}\ , \tag{10}$$

where (θ', D) is a computable unifier of E, E' with respect to r and none of $C_1, \ldots, C_n$ contains an answer literal. We obtain the following result.

Superposition (Sup):

$$\frac{\underline{s \simeq t} \lor C \lor \mathtt{ans}(r) \quad \underline{L[s']} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor L[t] \lor C \lor C' \lor \mathtt{ans}(\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{s \simeq t} \lor C \lor \mathtt{ans}(r) \quad \underline{L[s']} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor L[t] \lor C \lor C' \lor \mathtt{ans}(r))\theta}$$

$$\frac{\underline{s \simeq t} \lor C \lor \mathtt{ans}(r) \quad \underline{u[s'] \not\simeq u'} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor u[t] \not\simeq u' \lor C \lor C' \lor \mathtt{ans}(\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{s \simeq t} \lor C \lor \mathtt{ans}(r) \quad \underline{u[s'] \simeq u'} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor u[t] \simeq u' \lor C \lor C' \lor \mathtt{ans}(r))\theta}$$

$$\frac{\underline{s \simeq t} \lor C \lor \mathtt{ans}(r) \quad \underline{u[s'] \simeq u'} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor u[t] \simeq u' \lor C \lor C' \lor \mathtt{ans}(\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{s \simeq t} \lor C \lor \mathtt{ans}(r) \quad \underline{u[s'] \not\simeq u'} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor u[t] \not\simeq u' \lor C \lor C' \lor \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' w.r.t. the argument of the answer literal in the rule conclusion (i.e. $\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r$ for the left-column rules, and r for the others); (rules on the first line only) $L[s']$ is not an equality literal; and (rules on the second and third line only) $u'\theta \not\succeq u[s']\theta$.

Binary resolution (BR):

$$\frac{\underline{A} \lor C \lor \mathtt{ans}(r) \quad \neg\underline{A'} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor C \lor C' \lor \mathtt{ans}(\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{A} \lor C \lor \mathtt{ans}(r) \quad \neg\underline{A'} \lor C' \lor \mathtt{ans}(r')}{(\underline{D} \lor \underline{r \not\simeq r'} \lor C \lor C' \lor \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of A, A' w.r.t. (first rule) $\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r$ or (second rule) r.

Factoring (F):

$$\frac{\underline{A} \lor \underline{A'} \lor C \lor \mathtt{ans}(r)}{(\underline{D} \lor A \lor C \lor \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of A, A' w.r.t. r.

Equality resolution (ER):

$$\frac{\underline{s \not\simeq t} \lor C \lor \mathtt{ans}(r)}{(\underline{D} \lor C \lor \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, t w.r.t. r.

Equality factoring (EF):

$$\frac{\underline{s \simeq t} \lor \underline{s' \simeq t'} \lor C \lor \mathtt{ans}(r)}{(\underline{D} \lor s \simeq t \lor t \not\simeq t' \lor C \lor \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' w.r.t. r; $t\theta \not\succeq s\theta$; and $t'\theta \not\succeq t\theta$.

Fig. 3. Selected rules of the extended superposition calculus $\mathbb{S}\mathrm{up}$ for reasoning with answer literals, with underlined literals being selected.

Lemma 4 [Soundness of Inferences with Answer Literals] *If the rule* (9) *is sound, the rules* (10) *are sound as well.*

We note that we keep the original rule (9) in $\mathcal{I}$, but impose that none of its premises $C_1, \ldots, C_n$ contains an answer literal. Clearly, neither the such modified rule (9) nor the new rules (10) introduce uncomputable symbols into answer literals. Rather, these rules add disequality constraints D into their conclusions and immediately select D for further applications of inference rules. Such a selection guides the saturation process in Algorithm 1 to first discharge the constraints D containing uncomputable symbols with the aim of deriving a clause $C' \lor \mathtt{ans}(r')$ where C' is computable. The clause $C' \lor \mathtt{ans}(r')$ is then converted into a program with conditions using Corollary 2.

Superposition with answer literals. We make the inference rule modifications (9), together with the addition of new rules (10), for each inference rule of the $\mathbb{S}$up calculus from Figure 1. Further, we also ensure that rules with multiple premises, when applied on several premises containing answer literals, *derive clauses with at most one answer literal*. We therefore introduce the following two rule modifications. (i) We use the $\texttt{if}-\texttt{then}-\texttt{else}$ constructor to combine answer literals of premises, by adapting the use of $\texttt{if}-\texttt{then}-\texttt{else}$ within binary resolution [12,13,21] to superposition rules. (ii) We use an answer literal from only one of the rule premises in the rule conclusion and add new disequality constraint $r \not\simeq r'$ between the premises' answer literal arguments, similar to the constraints D of the computable unifier. Analogously to the computable unifier constraints, we immediately select this disequality constraint $r \not\simeq r'$.

The resulting extension of the $\mathbb{S}$up calculus with answer literals is given in Figure 3. In addition to the rules of Figure 3, the extended calculus contains rules constructed as (10) for superposition and binary resolution rules of Figure 1. Using Lemma 4, we conclude the following.

Lemma 5 [Soundness of $\mathbb{S}$up with Answer Literals] *The inference rules from Figure 3 of the extended $\mathbb{S}$up calculus with answer literals are sound.*

By the soundness results of Lemmas 4–5, Corollaries 2–3 imply that, when applying the calculus of Figure 3 in the saturation-based program synthesis approach of Algorithm 1, we construct correct programs.

Example 2. We illustrate the use of Algorithm 1 with the extended $\mathbb{S}$up calculus of Figure 3, strengthening our motivation from Section 3 with $\texttt{if}-\texttt{then}-\texttt{else}$ reasoning. To this end, consider the functional specification over group theory:

$$\forall x, y. \exists z. (x * y \not\simeq y * x \rightarrow z * z \not\simeq e), \tag{11}$$

asserting that, if the group is not commutative, there is an element whose square is not e. In addition to the axioms (A1)-(A3) of Figure 2, we also use the right identity axiom (A2') $\forall x.\ x * e \simeq x$.[7] Based on Algorithm 1, we obtain the following derivation of the program for (11):

1. $\sigma_1 * \sigma_2 \not\simeq \sigma_2 * \sigma_1 \vee \texttt{ans}(z)$ [preprocessed specification]
2. $e \simeq z * z \vee \texttt{ans}(z)$ [preprocessed specification]
3. $\sigma_1 * \sigma_2 \not\simeq \sigma_2 * \sigma_1$ [answer literal removal 1. (Algorithm 1, line 10)]
4. $x * (x * y) \simeq e * y \vee \texttt{ans}(x)$ [Sup 2., A3]
5. $e \simeq x * (y * (x * y)) \vee \texttt{ans}(x * y)$ [Sup A3, 2.]
6. $x * (x * y) \simeq y \vee \texttt{ans}(x)$ [Sup 4., A2]
7. $x * e \simeq y * (x * y) \vee \texttt{ans}(\texttt{if } e \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y)$ [Sup 6., 5.]
8. $y * (x * y) \simeq x \vee \texttt{ans}(\texttt{if } e \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y)$ [Sup 7., A2']
9. $x*y \simeq y*x \vee \texttt{ans}(\texttt{if } x*(y*x) \simeq y \texttt{ then } x \texttt{ else if } e \simeq x*(y*(x*y)) \texttt{ then } x \texttt{ else } x*y)$ [Sup 6., 8.]
10. $\texttt{ans}(\texttt{if } \sigma_1*(\sigma_2*\sigma_1) \simeq \sigma_2 \texttt{ then } \sigma_1 \texttt{ else if } e \simeq \sigma_1*(\sigma_2*(\sigma_1*\sigma_2)) \texttt{ then } \sigma_1 \texttt{ else } \sigma_1*\sigma_2)$ [BR 9., 3.]
11. $\square$ [answer literal removal 11. (Algorithm 1, line 10)]

[7] We include axiom (A2') only to shorten the presentation of the obtained derivation.

The programs with conditions collected during saturation-based program synthesis, in particular corresponding to steps 3. and 11. above, are:

$P_1[x, y] := \langle z, x * y \simeq y * x\rangle$
$P_2[x, y] := \langle \texttt{if } x * (y * x) \simeq y \texttt{ then } x \texttt{ else } (\texttt{if } e \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y),$
$x * y \not\simeq y * x\rangle$

Note the variable z, representing an arbitrary witness, in $P_1[x, y]$. An arbitrary value is a correct witness in case $x*y \simeq y*x$ holds, as in this case (11) is trivially satisfied. Thus, we do not need to consider the case $x*y \simeq y*x$ separately. Hence, we construct the final program $P[x, y]$ only from $P_2[x, y]$ and obtain:

$$P[x, y] := \texttt{if } x*(y*x)\simeq x \texttt{ then } x \texttt{ else } (\texttt{if } e\simeq x*(y*(x*y)) \texttt{ then } x \texttt{ else } x*y)$$

We conclude this section by illustrating the benefits of computable unifiers.

Example 3. Consider the group theory specification

$$\forall x, y.\exists z.\ z * (i(x) * i(y)) = e, \tag{12}$$

describing the inverse element z of $i(x) * i(y)$. We annotate the inverse $i(\cdot)$ as uncomputable to disallow the trivial solution $i(i(x) * i(y))$. Using computable unifiers, we synthesize[8] the program $y * x$; that is, a program computing $y * x$ as the inverse of $i(x) * i(y)$.

6 Computable Unification with Abstraction

When compared to the $\mathbb{S}$up calculus of Figure 1, our extended $\mathbb{S}$up calculus with answer literals from Figure 3 uses computable unifiers instead of mgus. To find computable unifiers, we introduce Algorithm 2 by extending a standard unification algorithm [17,7] and an algorithm for unification with abstraction of [16]. Algorithm 2 combines computable unifiers with mgu computation, resulting in the computable unifier $\theta := \texttt{mgu}_{\texttt{comp}}(E_1, E_2, E_3)$ to be further used in Figure 3.

Algorithm 2 modifies a standard unification algorithm to ensure computability of $E_3\theta$. Changes compared to a standard unification algorithm are highlighted. Algorithm 2 does not add $s \mapsto t$ to θ if s is a variable in E_3 and t is uncomputable. Instead, if t is $f(t_1, \ldots, t_n)$ where f is computable but not all $t_1, \ldots, t_n$ are computable, we extend θ by $s \mapsto f(x_1, \ldots, x_n)$ and then add equations $x_1 = t_1, \ldots, x_n = t_n$ to the set of equations $\mathcal{E}$ to be processed. Otherwise, f is uncomputable and we perform an abstraction: we consider s and t to be unified under the condition that $s \simeq t$ holds. Therefore we add a constraint $s \not\simeq t$ to the set of literals $\mathcal{D}$ which will be added to any clause invoking the computable unifier. To discharge the literal $s \not\simeq t$, one must prove $s \simeq t$. While s can be later substituted for other terms, as long as we use $\texttt{mgu}_{\texttt{comp}}$, s will never be substituted for an uncomputable term. Thus, we conclude the following result.

Theorem 6. *Let E_1, E_2, E_3 be expressions. Then $(\theta, D) := \texttt{mgu}_{\texttt{comp}}(E_1, E_2, E_3)$ is a computable unifier.*

[8] see derivation in Appendix B

Algorithm 2 Computable Unification with Abstraction

function $\mathtt{mgu}_{\mathtt{comp}}(E_1, E_2, E_3)$
  if E_3 is uncomputable **then** fail
  let $\mathcal{E}$ be a set of equations and θ be a substitution; $\mathcal{E} := \{E_1 = E_2\}$; $\theta := \{\}$
  let $\mathcal{D}$ be a set of disequalities; $\mathcal{D} := \emptyset$
  repeat
    if $\mathcal{E}$ is empty **then**
      return (θ, D) where D is the disjunction of literals in $\mathcal{D}$
    Select an equation $s = t$ in $\mathcal{E}$ and remove it from $\mathcal{E}$
    if s coincides with t **then** do nothing
    else if s is a variable and s does not occur in t **then**
      if s does not occur in E_3 or t is computable **then** $\theta := \theta \circ \{s \mapsto t\}; \mathcal{E} = \mathcal{E}\{s \mapsto t\}$
      else if $t = f(t_1, \ldots, t_n)$ and f is computable **then**
        $\theta := \theta \circ \{s \mapsto f(x_1, \ldots, x_n)\};\ \mathcal{E} := \mathcal{E}\{s \mapsto f(x_1, \ldots, x_n)\} \cup \{x_1 = t_1, \ldots, x_n = t_n\}$
          where $x_1, \ldots, x_n$ are fresh variables
      else if $t = f(t_1, \ldots, t_n)$ and f is uncomputable **then** $\mathcal{D} := \mathcal{D} \cup \{s \not\simeq t\}$
    else if s is a variable and s occurs in t **then** fail
    else if t is a variable **then** $\mathcal{E} := \mathcal{E} \cup \{t = s\}$
    else if s and t have different top-level symbols **then** fail
    else if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ **then** $\mathcal{E} := \mathcal{E} \cup \{s_1 = t_1, \ldots, s_n = t_n\}$

7 Implementation and Experiments

Implementation. We implemented our saturation-based program synthesis approach in the Vampire prover [10]. We used Algorithm 1 with the extended $\mathbb{S}$up calculus of Figure 3. The implementation, consisting of approximately 1100 lines of C++ code, is available at https://github.com/vprover/vampire/tree/synthesis. The synthesis functionality can be turned on using the option `--question_answering synthesis`.

Vampire accepts functional specifications in an extension of the SMT-LIB2 format [4], by using the new command `assert-not` to mark the specification. We consider interpreted theory symbols to be computable. Uninterpreted symbols can be annotated as uncomputable via the command `(set-option :uncomputable (symbol1 ... symbolN))`.

Our implementation also integrates Algorithm 1 with the AVATAR architecture [25].[9] We modified the AVATAR framework to only allow splitting over ground computable clauses that do not contain answer literals. Further, if we derive a clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ with AVATAR assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$, where $C[\overline{\sigma}]$ is ground and computable, we replace it by the clause $C[\overline{\sigma}] \vee \bigvee_{i=1}^{m} \neg C_i[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ without any assertions. We then immediately record a program with conditions $\langle r[\overline{x}], \neg C[\overline{x}] \wedge \bigwedge_{i=1}^{m} C_i[\overline{x}] \rangle$, and replace the clause by $C[\overline{\sigma}] \vee \bigvee_{i=1}^{m} \neg C_i[\overline{\sigma}]$ (see lines 7-10 of Algorithm 1), which may be then further split by AVATAR.

Finally, our implementation simplifies the programs we synthesize. If during Algorithm 1 we record a program $\langle z, F \rangle$ where z is a variable, we do not use

[9] We include a short description of AVATAR in Appendix C.

this program in the final program construction (line 12 of Algorithm 1) even if F occurs in the derivation of $\square$ (see Example 2).

Examples and experimental setup. The goal of our experimental evaluation is to showcase the benefits of our approach on problems that are deemed to be hard, even unsolvable, by state-of-the-art synthesis techniques. We therefore focused on first–order theory reasoning and evaluated our work on the group theory problems of Examples 1-3, as well as on integer arithmetic problems.

As the SMT-LIB2 format can easily be translated into the SyGuS 2.1 syntax [15], we compared our results to cvc5 1.0.4 [3], supporting SyGuS-based synthesis [2]. Our experiments were run on an AMD Epyc 7502, 2.5 GHz CPU with 1 TB RAM, using a 5 minutes time limit per example. Our benchmarks as well as the configurations for our experiments are available at: https://github.com/vprover/vampire_benchmarks/tree/master/synthesis

Experimental results with group theory properties. VAMPIRE synthesizes the solutions of the Examples 1-3 in 0.01, 13, and 0.03 seconds, respectively. Since these examples use uninterpreted functions, they cannot be encoded in the SyGuS 2.1 syntax, showcasing the limits of other synthesis tools.

Experimental results with maximum of $n \geq 2$ integers. For the maximum of 2 integers, the specification is $\forall x_1, x_2 \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\big(y \geq x_1 \wedge y \geq x_2 \wedge (y = x_1 \vee y = x_2)\big)$, and the program we synthesize is `if` $x_1 < x_2$ `then` x_2 `else` x_1. Both our work and cvc5 are able to synthesize programs choosing the maximal value for up to $n = 23$ input variables, as summarized below. For $n > 23$, both VAMPIRE and cvc5 time out.

Number n of variables for which max is synthesized	2	5	10	15	20	22	23
VAMPIRE	0.03	0.03	0.05	1	13	55	215
cvc5	0.01	0.03	0.6	6.8	88	188	257

Experimental results with polynomial equations. VAMPIRE can synthesize the solution of polynomial equations; for example, for $\forall x_1, x_2 \in \mathbb{Z}.\exists y \in \mathbb{Z}.(y^2 = x_1^2 + 2x_1x_2 + x_2^2)$, we synthesize $x_1 + x_2$. VAMPIRE finds the corresponding program in 26 seconds using simple first-order reasoning, while cvc5 fails in our setup.

8 Related Work

Our work builds upon deductive synthesis [13] adapted for the resolution calculus [12,21]. We extend this line of work with saturation-based program synthesis, by using adjustments of the superposition calculus.

Component-based synthesis of recursion-free programs [20] from logical specifications is addressed in [20,6,23]. The work of [20] uses first-order theorem proving to prove specifications and extract programs from proofs. In [6,23], $\exists\forall$ formulas are produced to capture specifications over component properties and SMT solving is applied to find a term satisfying the formula, corresponding to a straight-line program. We complement [20] with saturation-based superposition proving and avoid template-based SMT solving from [6,23].

A prominent line of research comes with syntax guided synthesis (SyGuS) [1], where functional specifications are given using a context-free grammar. This grammar yields program templates to be synthesized via an enumerative search procedure based on SMT solving [3,8]. We believe our work is complementary to SyGuS, by strengthening first-order reasoning for program synthesis, as evidenced by Examples 1–3.

The sketching technique [18,24] synthesizes program assignments to variables, using an alternative framework to the program synthesis setting we rely upon. In particular, sketching addresses domains that do not involve input logical formulas as functional specifications, such as example-guided synthesis [22].

9 Conclusions

We extend saturation-based proof search to saturation-based program synthesis, aiming to derive recursion-free programs from specifications. We integrate answer literals with saturation, and modify the superposition calculus and unification to synthesize computable programs. Our initial experiments show that a first-order theorem prover becomes an efficient program synthesizer, potentially opening up interesting avenues toward recursive program synthesis, for example using saturation-based proving with induction.

Acknowledgements. We thank Haniel Barbosa for support with experiments with cvc5. This work was partially funded by the ERC CoG ARTIST 101002685, and the FWF grants LogiCS W1255-N23 and LOCOTES P 35787.

References

1. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-Guided Synthesis. In: Dependable Software Systems Engineering, pp. 1–25 (2015)
2. Alur, R., Fisman, D., Padhi, S., Reynolds, A., Singh, R., Udupa, A.: SyGuS-Comp 2019. `https://sygus.org/comp/2019/` (2019)
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: TACAS. pp. 415–442 (2022)
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard Version 2.6. `www.SMT-LIB.org` (2021)
5. Green, C.: Theorem-Proving by Resolution as a Basis for Question-Answering Systems. Machine Intelligence **4**, 183–205 (1969)
6. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of Loop-Free Programs. In: PLDI. p. 62–73 (2011)
7. Hoder, K., Voronkov, A.: Comparing Unification Algorithms in First-Order Theorem Proving. In: KI. pp. 435–443 (2009)

8. Jha, S., Seshia, S.A.: A Theory of Formal Synthesis via Inductive Learning. Acta Informatica **54**(7), 693–726 (2017)
9. Korovin, K., Kovacs, L., Reger, G., Schoisswohl, J., Voronkov, A.: ALASCA: Reasoning in Quantified Linear Arithmetic. In: TACAS (2023), to appear
10. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV. pp. 1–35 (2013)
11. Kunen, K.: The Semantics of Answer Literals. J. of Automated Reasoning **17**(1), 83–95 (1996)
12. Lee, R.C.T., Waldinger, R.J., Chang, C.L.: An Improved Program-Synthesizing Algorithm and Its Correctness. Commun. ACM (4), 211–217 (1974)
13. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
14. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Handbook of Automated Reasonings, vol. I, pp. 371–443. Elsevier and MIT Press (2001)
15. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.1. `https://sygus.org/language/` (2021)
16. Reger, G., Suda, M., Voronkov, A.: Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning. In: TACAS. pp. 3–22 (2018)
17. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM **12**(1), 23–41 (1965)
18. Solar-Lezama, A.: The Sketching Approach to Program Synthesis. In: APLAS. pp. 4–13 (2009)
19. Srivastava, S., Gulwani, S., Foster, J.S.: From Program Verification to Program Synthesis. In: POPL. p. 313–326 (2010)
20. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I.: Deductive Composition of Astronomical Software from Subroutine Libraries. In: CADE. pp. 341–355 (1994)
21. Tammet, T.: Completeness of Resolution for Definite Answers. J. of Logic and Computation **5**(4), 449–471 (08 1995)
22. Thakkar, A., Naik, A., Sands, N., Alur, R., Naik, M., Raghothaman, M.: Example-Guided Synthesis of Relational Queries. In: PLDI. p. 1110–1125 (2021)
23. Tiwari, A., Gascón, A., Dutertre, B.: Program Synthesis Using Dual Interpretation. In: CADE. pp. 482–497 (2015)
24. Torlak, E., Bodik, R.: Growing Solver-Aided Languages with Rosette (2013)
25. Voronkov, A.: AVATAR: The Architecture for First-Order Theorem Provers. In: CAV. pp. 696–710 (2014)

A Proofs

In the following we use the notion of *universal closure* of a formula F, which is the formula $\forall\overline{z}.F$, where $\overline{z}$ are all free variables of F.

Theorem 1 [Semantics of Clauses with Answer Literals] *Let C be a clause not containing an answer literal. Assume that, using a saturation algorithm based on a sound inference system $\mathcal{I}$, the clause $C \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived from the set of clauses consisting of initial assumptions $A_1, \ldots, A_n$, the clausified formula $\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y))$ and additional assumptions $C_1, \ldots, C_m$. Then,*

$$A_1, \ldots, A_n, C_1, \ldots, C_m \vdash C, F[\overline{\sigma}, r[\overline{\sigma}]].$$

That is, under the assumptions $C_1, \ldots, C_m, \neg C$, the computable term $r[\overline{\sigma}]$ provides a definite answer to (6).

Proof. We consider the calculus which was used for deriving $C \vee \mathtt{ans}(r[\overline{\sigma}])$, but with lifted ordering and selection constraints. Since the soundness of the calculus does not depend on these constraints, the calculus without the constraints is sound as well. Now, since $\mathtt{ans}$ is uninterpreted, we can replace $\mathtt{ans}(y)$ by $y \not\simeq r[\overline{\sigma}]$, and obtain a derivation of $C \vee r[\overline{\sigma}] \not\simeq r[\overline{\sigma}]$ from $A_1, \ldots, A_n, \forall y.\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$ using the calculus without the constraints.[10]

We want to show that

$$\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{m} C_i \rightarrow C \vee F[\overline{\sigma}, r[\overline{\sigma}]] \tag{13}$$

is valid. Hence, we need to show that in each interpretation, in which the antecedent is true, also the consequent is true. Let us consider such an interpretation I. We distinguish two cases. First, assume that $\forall y.\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$ is true in I. Then since all assumptions from which we derived $C \vee r[\overline{\sigma}] \not\simeq r[\overline{\sigma}]$ are true in I and since the inference system is sound, also $C \vee r[\overline{\sigma}] \not\simeq r[\overline{\sigma}]$ is true. That clause is equivalent to C, hence C is true, which makes the consequent of (13) true. Second, assume that $\forall y.\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$ is false in I. Then its negation, $\neg\forall y.\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee y \not\simeq r[\overline{\sigma}])$, equivalent to $\exists y.(F[\overline{\sigma}, y] \wedge y \simeq r[\overline{\sigma}])$, equivalent to $F[\overline{\sigma}, r[\overline{\sigma}]]$ must be true in I. Hence, the consequent of (13) is true also in this case. Therefore (13) is valid. □

Corollary 2 [Programs with Conditions] *Let $r[\overline{\sigma}]$ be a computable term and $C[\overline{\sigma}]$ a ground computable clause not containing an answer literal. Assume that clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived from the set of initial clauses $A_1, \ldots, A_n$, the clausified formula $\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y))$ and additional ground computable*

[10] The derivation might not have been possible in the calculus with the ordering and selection constraints due to replacing the positive literal $\mathtt{ans}(y)$ with the negative literal $y \not\simeq r[\overline{\sigma}]$ containing different symbols.

assumptions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$*, by using saturation based on a sound inference system* $\mathcal{I}$*. Then,*

$$\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \land \neg C[\overline{x}]\rangle$$

is a program with conditions for (6)*.*

Proof. From Theorem 1 follows that $\bigwedge_{i=1}^{n} A_i \land \bigwedge_{i=1}^{m} C_i[\overline{\sigma}] \rightarrow C[\overline{\sigma}] \lor F[\overline{\sigma}, r[\overline{\sigma}]]$ holds. Since $\overline{\sigma}$ are fresh uninterpreted constants, we obtain that $\bigwedge_{i=1}^{n} A_i \land \bigwedge_{i=1}^{m} C_i[\overline{x}] \rightarrow C[\overline{x}] \lor F[\overline{x}, r[\overline{x}]]$ is valid as well, and that is equivalent to $\bigwedge_{j=1}^{m} C_j[\overline{x}] \land \neg C[\overline{x}] \rightarrow (\bigwedge_{i=1}^{n} A_i \rightarrow F[\overline{x}, r[\overline{x}]])$. Therefore $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \land \neg C[\overline{x}]\rangle$ is a program with conditions for $A_1 \land \ldots \land A_n \rightarrow \forall \overline{x}. \exists y. F[\overline{x}, y]$. □

Corollary 3 [From Programs with Conditions to Programs for (6)**]** *Let* $P_1[\overline{x}], \ldots, P_k[\overline{x}]$*, where* $P_i[\overline{x}] = \langle r_i[\overline{x}], \bigwedge_{j=1}^{i-1} C_j[\overline{x}] \land \neg C_i[\overline{x}]\rangle$*, be programs with conditions for* (6)*, such that* $\bigwedge_{i=1}^{n} A_i \land \bigwedge_{i=1}^{k} C_i[\overline{x}]$ *is unsatisfiable. Then* $P[\overline{x}]$*, given by*

$$\begin{aligned} P[\overline{x}] := {} & \texttt{if } \neg C_1[\overline{x}] \texttt{ then } r_1[\overline{x}] \\ & \texttt{else if } \neg C_2[\overline{x}] \texttt{ then } r_2[\overline{x}] \\ & \quad \ldots \\ & \texttt{else if } \neg C_{k-1}[\overline{x}] \texttt{ then } r_{k-1}[\overline{x}] \\ & \texttt{else } r_k[\overline{x}], \end{aligned} \tag{8}$$

is a program for (6)*.*

Proof. For any interpretation I and any variable assignment v, let p be the smallest index such that $\neg C_p[\overline{x}]$ holds in I under v, but all $\neg C_j[\overline{x}]$, where $1 \leq j < p$, do not hold in I under v. Since $\bigwedge_{i=1}^{n} A_i \land \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable, under the assumptions $A_1, \ldots, A_n$ such a p has to exist. Then in I under v and under the assumptions $A_1, \ldots, A_n$, the interpretation of $P[\overline{x}]$ is the same as the interpretation of $r_p[\overline{x}]$.

Further, since $\bigwedge_{j=1}^{p-1} C_j[\overline{x}] \land \neg C_p[\overline{x}]$ is the condition for $P_p[\overline{x}]$, from the definition of a program with conditions we obtain that $A_1 \land \cdots \land A_n \rightarrow F[\overline{x}, r_p[\overline{x}]]$ holds in I under v. Hence also $A_1 \land \cdots \land A_n \rightarrow F[\overline{x}, P[\overline{x}]]$ holds in I under v.

Finally, since this argument holds for any I and v, and since all $A_1, \ldots, A_n$ are closed formulas, also $A_1 \land \cdots \land A_n \rightarrow \forall \overline{x}. F[\overline{x}, P[\overline{x}]]$ holds. Therefore $P[\overline{x}]$ is a program for (6). □

Lemma 4 [Soundness of Inferences with Answer Literals] *If the rule* (9) *is sound, the rules* (10) *are sound as well.*

Proof. For clarity we repeat the original rule:

$$\frac{C_1 \quad \cdots \quad C_n}{C\theta}\ , \tag{9}$$

where θ is a substitution such that $E\theta \simeq E'\theta$ holds for some expressions E, E'.

We will prove the soundness of the new rule

$$\frac{C_1 \vee \texttt{ans}(r) \quad C_2 \quad \cdots \quad C_n}{(D \vee C \vee \texttt{ans}(r))\theta'} \quad , \tag{10'}$$

where (θ', D) is a computable unifier of E, E' with respect to r, and none of $C_1, \ldots, C_n$ contains an answer literal. The proof of soundness of the other new rules of (10) is analogous.

Assume interpretation I to be a model of the universal closures of the premises of (10'), but not a model of the universal closure of its conclusion. Then $D\theta', C\theta'$ and $\texttt{ans}(r)\theta'$ are false in I. From $D\theta'$ being false in I and from (θ', D) being an abstract unifier follows that $E\theta' \simeq E'\theta'$ holds. We can therefore set $\theta := \theta'$. From the soundness of (9) and $C\theta'$ being false in I then follows that some of $C_1, \ldots, C_n$ is false in I. However, none of $C_2, \ldots, C_n$ can be false in I, because we assumed all premises of (10') to be true in I. Hence, C_1 is false in I. Further, from $\texttt{ans}(r)\theta'$ being false in I follows that $\texttt{ans}(r)$ is false in I. However, that means that $C_1 \vee \texttt{ans}(r)$ is false in I, which contradicts the assumption that the universal closures of all premises of rule (10') are true in I.

Hence, the rule (10') is sound. □

Lemma 5 [Soundness of $\mathbb{S}$up with Answer Literals] *The inference rules from Figure 3 of the extended $\mathbb{S}$up calculus with answer literals are sound.*

Proof. Soundness of the factoring, equality factoring and equality resolution rules follows from Lemma 4.

We will prove soundness for the first superposition rule and the second binary resolution rule. The proofs for other superposition and binary resolution rules are analogous.

For clarity we repeat the first superposition rule of Figure 3:

$$\frac{s \simeq t \vee C \vee \texttt{ans}(r) \quad L[s'] \vee C' \vee \texttt{ans}(r')}{(D \vee L[t] \vee C \vee C' \vee \texttt{ans}(\texttt{if } s \simeq t \texttt{ then } r' \texttt{ else } r))\theta}$$

Assume interpretation I to be a model of the universal closures of the premises of the rule, but not a model of the universal closure of its conclusion. Then there is some variable assignment v such that $(D \vee L[t] \vee C \vee C' \vee \texttt{ans}(\texttt{if } s \simeq t \texttt{ then } r' \texttt{ else } r))\theta$ is false in I under v. Let v' be a variable assignment that assigns to each variable x the value that $x\theta$ has in I under v. Then:

1. From $L[t]\theta, C\theta, C'\theta$ being false in I under v follows that $L[t], C, C'$ are false in I under v'.
2. Since $D\theta$ is false in I under v, from (θ, D) being an abstract unifier of s, s' follows that $s\theta \simeq s'\theta$ is true in I under v, and therefore s, s' have the same interpretation in I under v'. Then consider two cases:
   (a) $s \simeq t$ is true in I under v' and $s\theta \simeq t\theta$ is true in I under v. Then from $\texttt{ans}(\texttt{if } s \simeq t \texttt{ then } r' \texttt{ else } r)\theta$ being false in I under v follows that

$\mathtt{ans}(r')\theta$ is false in I under v and therefore $\mathtt{ans}(r')$ is false in I under v'. Also from $s \simeq t$ being true in I under v', 1., and 2. follows that $L[s']$ is false in I under v'. Then the whole second premise of the rule is false in I under v', which is a contradiction with the assumption that I is a model of its universal closure.

(b) $s \simeq t$ is false in I under v' and $s\theta \simeq t\theta$ is false in I under v. This case leads similarly to the first premise being false, in contradiction with the assumption.

Therefore the first superposition rule is sound.

For clarity we repeat the second binary resolution rule of Figure 3:

$$\frac{A \vee C \vee \mathtt{ans}(r) \quad \neg A' \vee C' \vee \mathtt{ans}(r')}{(D \vee r \not\simeq r' \vee C \vee C' \vee \mathtt{ans}(r))\theta}$$

Assume interpretation I to be a model of the universal closures of the premises of the rule, but not a model of the universal closure of its conclusion. Then there is some variable assignment v such that $(D \vee r \not\simeq r' \vee C \vee C' \vee \mathtt{ans}(r))\theta$ is false in I under v. Let v' be a variable assignment that assigns to each variable x the value that $x\theta$ has in I under v. Then:

1. From $r\theta \not\simeq r'\theta, C\theta, C'\theta$ being false in I under v follows that $r \not\simeq r', C, C'$ are false in I under v'. Therefore r, r' have the same interpretation in I under v'.
2. Since $\mathtt{ans}(r)\theta$ is false in I under v, also $\mathtt{ans}(r)$ is false in I under v'. Then from 1. follows that $\mathtt{ans}(r')$ is also false in I under v'.
3. Since $D\theta$ is false in I under v, from (θ, D) being an abstract unifier of A, A' follows that $A\theta, A'\theta$ have the same interpretation in I under v, and therefore A, A' have the same interpretation in I under v'. Therefore, only one of $A, \neg A'$ is true in I under v', which together with $C, C', \mathtt{ans}(r), \mathtt{ans}(r')$ being false in I under v' forms a contradiction with the assumption that I is a model of both premises of the rule.

Therefore the second binary resolution rule is sound as well. □

Theorem 6. *Let E_1, E_2, E_3 be expressions. Then $(\theta, D) := \mathtt{mgu}_{\mathtt{comp}}(E_1, E_2, E_3)$ is a computable unifier.*

Proof. We will denote the subexpression of the expression E at position p by $E|p$.

We first prove that (θ, D) is an abstract unifier of E_1, E_2. If $E_1\theta|p'$ and $E_2\theta|p'$ differ, there has to be a position p, where p' is a prefix of p, such that the top-level symbol of $E_1\theta|p$ and $E_2\theta|p$ differs. From the construction of θ follows that for any position p, the subexpressions $E_1\theta|p, E_2\theta|p$ differ in their top-level symbol only if $E_1|p = s$ and $E_2|p = f(t_1, \dots, t_n)$ (or, symmetrically, $E_1|p = f(t_1, \dots, t_n)$ and $E_2|p = s$) where s is a variable and f is uncomputable. However, in this case $s \not\simeq f(t_1, \dots, t_n)$ occurs in D. Therefore, for any interpretation I, any variable assignment v, and any position p', the interpretations of $E_1\theta|p', E_2\theta|p'$ in I under

v will either be the same, or $s\theta \not\simeq f(t_1, \ldots, t_n)\theta$ will be true in I under v. Hence, $(D \vee E_1 \simeq E_2)\theta$ is valid, and therefore (θ, D) is an abstract unifier of E_1, E_2.

Next, we prove that $E_3\theta$ is computable. Since the algorithm successfully terminated, E_3 must have been computable (otherwise it would fail). Further, the algorithm only extends the substitution θ by $s \mapsto t$ where t is uncomputable if s does not occur in E_3. Thus, $E_3\theta$ is computable, and hence (θ, D) is a computable unifier.

B Example

A detailed version of Example 3 follows.

Example 3. Consider the group axioms (A1)-(A3) of Figure 2, the additional axioms (A1') $\forall x.\ x * i(x) \simeq e$ for right inverse and (A2') $\forall x.\ x * e \simeq x$ for right identity (symmetric to (A1), (A2)),[11] and the following specification

$$\forall x, y. \exists z.\ z * (i(x) * i(y)) = e, \tag{14}$$

describing the inverse element of $i(x) * i(y)$. The trivial program derivation for this specification would only have three steps:

1.	$e \not\simeq x * (i(\sigma_1) * i(\sigma_2)) \vee \mathtt{ans}(x)$	[preprocessed specification]
2.	$\mathtt{ans}(i(i(\sigma_1) * i(\sigma_2)))$	[BR A1, 1.]
3.	$\square$	[answer literal removal 2.]

To disallow the trivial solution, $i(i(x)*i(y))$, we annotate the function symbol i as uncomputable. Therefore we do not perform the step 2. from above, but instead perform binary resolution with the computable unifier $(\{x \mapsto i(i(\sigma_1) * i(\sigma_2))\}, x \not\simeq i(i(\sigma_1) * i(\sigma_2)))$, leading to the following derivation:

1.	$e \not\simeq x * (i(\sigma_1) * i(\sigma_2)) \vee \mathtt{ans}(x)$	[preprocessed specification]
2.	$i(i(\sigma_1) * i(\sigma_2)) \not\simeq x \vee \mathtt{ans}(x)$	[BR A1, 1.]
3.	$i(x) * (x * y) \simeq e * y$	[Sup A3, A1]
4.	$i(x) * (x * y) \simeq y$	[Sup 3., A2]
5.	$i(i(x)) * e \simeq x$	[Sup 4., A1]
6.	$i(i(x)) \simeq x$	[Sup 5., A2']
7.	$e \simeq x * (y * i(x * y))$	[Sup A1', A3]
8.	$x * i(y * x) \simeq i(x) * e$	[Sup 4., 7.]
9.	$i(x) = y * i(x * y)$	[Sup 8., A2']
10.	$i(x) * i(y) = i(y * x)$	[Sup 4., 9.]
11.	$i(i(\sigma_2, \sigma_1)) \not\simeq x \vee \mathtt{ans}(x)$	[Sup 10., 2.]
12.	$\mathtt{ans}(\sigma_2 * \sigma_1)$	[BR 6., 11.]
13.	$\square$	[answer literal removal 12.]

We synthesize the program $P[x, y] := y * x$. Note that there exists a different derivation only using computable unifiers in the form $(\theta, \square)$ (i.e., not using abstraction).

[11] We include the symmetric axioms to shorten the derivation for presentation purposes. The derivation would work also without (A1'), (A2').

C Splitting and AVATAR

One of the keys to the efficiency of saturation-based theorem proving is *clause splitting*, with the leading approach being the AVATAR architechture [25]. The main idea of splitting is as follows. Let S be a set of clauses and $C_1 \vee C_2$ a clause such that C_1, C_2 have disjoint sets of variables. We call such clauses C_1, C_2 the *components* of $C_1 \vee C_2$. Then $S \cup \{C_1 \vee C_2\}$ is unsatisfiable iff both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. Therefore, instead of checking satisfiability of a set of large clauses, we check the satisfiability of multiple sets of smaller clauses. AVATAR implements this idea by using an interplay between a saturation-based first-order theorem prover and a SAT/SMT solver. The SAT/SMT solver finds a set of clause components, satisfiability of which implies satisfiability of all split clauses. These components, called *assertions*, are then used by the theorem prover for further derivations in saturation. All clauses derived using assertions $C_1, \ldots, C_n$ are called *clauses with assertion* $C_1, \ldots, C_n$.

However, when using Algorithm 1 for program synthesis with (standard) AVATAR to saturate a preprocessed specification (7), we may derive a clause $\mathtt{ans}(r[\overline{\sigma}])$ with assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$. By Theorem 1, we then obtain

$$A_1, \ldots, A_n, C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}] \vdash F[\overline{\sigma}, r[\overline{\sigma}]].$$

If $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$ are computable and ground, then $\langle r[\overline{x}], \bigwedge_{i=1}^{m} C_i[\overline{x}] \rangle$ is a program with conditions. However, if not all of the assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$ are computable and ground, then Algorithm 1 should continue reasoning with these assertions with the aim of reducing them to computable and ground literals. This, however, is not directly possible in the AVATAR framework. Hence, we modify AVATAR as described in Section 7.