Completeness of Synthesis Under Realizability Assumptions Using Superposition

Márton Hajdu, Petra Hozzová, Laura Kovács, and Eva Maria Wagner(✉)

TU Wien, Vienna, Austria
eva.maria.wagner@tuwien.ac.at

Abstract. Program synthesis is the task of automatically deriving a program that has been specified by a user in advance. Combining automated theorem proving with program synthesis enables the automated construction of proven-to-be-correct programs, thereby ensuring software reliability. In this paper, we consider the superposition-based calculus extended to support synthesis of recursion-free programs allowing reasoning with uncomputable symbols. We present cases where the calculus fails and refine it to solve them. We prove that the refined calculus is sound. Finally, we also prove completeness in the following sense: if at least one computable program satisfying the given specification exists, we show that the modified calculus finds one.

Keywords: Program Synthesis · Saturation · Superposition · Theorem Proving

1 Introduction

Program synthesis focuses on constructing programs from a given functional specification. There are many different approaches for solving different flavors of this problem, including ones that guarantee that the found program satisfies a logical specification [14,15,24]; techniques that synthesize a program based on a set of input-output examples [8,10,26]; and LLM-based methods which require external correctness checking [18]. In this paper, we focus on an automated deductive synthesis approach using a superposition-based theorem prover, in which a saturation-based framework proves a given specification while simultaneously generating code conforming to that specification.

Recent work in [15] extracts a recursion-free program from a superposition-based proof of a logical specification (requirement) on the program. Our approach explores and revises this framework and solves *program synthesis in the presence of uncomputable symbols* [14,15]. Doing so, we impose the following syntactic restrictions on the program to be synthesized: (i) the program should use only so-called *computable* symbols, while (ii) its functional specification may use

A. Biere et al. (Eds.): IJCAR 2026, LNAI 16688, pp. 22–40, 2026.
https://doi.org/10.1007/978-3-032-32589-1_2

both computable and uncomputable symbols. Uncomputable symbols include, for example, symbols annotated as such by the user of the synthesis system, allowing for better control of what the output program should (not) use. Uncomputable symbols may, however, also include fresh symbols, such as Skolem functions, introduced during the proving process.

Motivating Example. We motivate our work using an example adapted from [12,22]. Consider the following constraints from the FLoC 2026 workshop schedule:

1. On Friday (f), the Vampire (v) workshop is taking place. Using the unary predicate w for workshops, we assert $\mathsf{w}(\mathsf{v})$.
2. On Saturday (s), the PAAR (p) workshop is taking place (thus, $\mathsf{w}(\mathsf{p})$).
3. Today (x) is either Friday (f) or Saturday (s).

Our task for program synthesis is to infer what workshop y takes place depending on x, with the condition that w is uncomputable, i.e. the answer should not contain w.

We encode this instance of program synthesis as follows. For readability, we might omit parentheses in unary symbol applications throughout the paper, e.g. we might write wv instead of $\mathsf{w}(\mathsf{v})$. We are looking for a program that is a witness for y in the following formula:

$$\varphi: \quad \forall x \exists y.(((x \approx \mathsf{f} \vee x \approx \mathsf{s}) \wedge (x \approx \mathsf{f} \rightarrow \mathsf{wv}) \wedge (x \approx \mathsf{s} \rightarrow \mathsf{wp})) \rightarrow \mathsf{w}y). \tag{1}$$

A program that is a witness for y in (1) is called a *solution of the synthesis problem* specified by (1); see Sect. 3 for precise formulation of program synthesis and its solution. In the case of (1), two possible programs found by our approach are **if** $x \approx \mathsf{f}$ **then** v **else** p and **if** $x \approx \mathsf{s}$ **then** p **else** v. ❑

Synthesis and Saturation. In our approach to program synthesis, we build on the saturation-based framework for program synthesis with uncomputable symbols using the superposition calculus, as introduced in [15]. While the approach was shown to be correct in [15], the question of completeness remained open until now. In this paper, we close this gap by investigating the *completeness of the calculus under the assumption of realizability*: if a computable program satisfying the specification exists, is the calculus guaranteed to derive it? We identify properties that make the calculus of [15] inherently incomplete. We adjust the calculus accordingly, resulting in our SUPRA framework (Sect. 5). We further prove completeness of SUPRA under the assumption of realizability (Sect. 5).

This paper starts with preliminaries in Sect. 2, summarizing necessary notions of first-order logic and automated reasoning. Section 3 defines our synthesis problem and presents a framework for solving it, by revising the setting of [15]. Following are *our main contributions*:

– We introduce a new superposition-based calculus for synthesis (Sect. 4). Our calculus is coined as Superposition with Realizability Assumptions, in short SUPRA. Motivated by examples that cannot be solved by [15], SUPRA includes tailored conditions for term orderings and selection functions.

– We prove completeness of our SUPRA calculus (Sect. 5). By completeness we mean that, if at least one computable program satisfying a given specification exists, our SUPRA calculus finds one such program.

We then review related work in Sect. 6, and conclude our paper in Sect. 7. The (omitted) proofs of our results from Sects. 4 and 5 are given in the extended version [11] of this paper.

2 Preliminaries

First-order Logic. We assume familiarity with standard multi-sorted first-order logic (FOL) with equality, where equality is denoted by $\approx$. We consider a fixed signature Σ consisting of a finite set of *function symbols* with associated arities and a set of *variables* $\mathcal{V}$; variables are not part of the signature. We define the set of *terms* $\mathcal{T}(\Sigma \cup \mathcal{V})$, or simply $\mathcal{T}$ when it is clear from the context, in the standard way over $\Sigma \cup \mathcal{V}$. We denote variables by x, y, z; terms by s, t, l, r, k; literals by L, K; clauses by C, D; and formulas by F, G, all possibly with indices. Further, we write α for Skolem constants. We denote lists of literals by $\mathcal{L}$ and $\mathcal{K}$. We denote the empty list of literals by ϵ. Let L be a literal and $\mathcal{L}$ a list of literals, then $L\mathcal{L}$ denotes the list of literals having L as a head and $\mathcal{L}$ as a tail. We write $\bar{a}$ for the tuple of variables or terms $a_1, \ldots, a_n$. When a term t is of the form $f(\bar{t})$, we say that f is the *top-level symbol* of t. By $\mathsf{cnf}(F)$ we denote the *clausal normal form* (CNF) of the formula F. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. We write $\hat{L}$ for the literal complementary to L. We write $t \not\approx s$ as a shorthand for $\neg(t \approx s)$. We use the symbol $\dot{\approx}$ to denote either $\approx$ or $\not\approx$. An *expression* is a term, literal, clause, or formula. We write $E_1 = E_2$ to denote syntactic equality of two expressions. If an expression does not contain variables, we call it *ground*. We write $E[t]$ to denote all (possibly zero) occurrences of the term t. Then, $E[s]$ denotes the expression $E[t]$ where all occurrences of t are replaced by the term s. Formulas with free variables are considered implicitly universally quantified; that is, we consider closed formulas. A *substitution* σ is a mapping from variables to terms such that the set $\{x \mid \sigma(x) \neq x\}$ of variables is finite. A substitution σ is a *unifier* of two expressions E and E' if $E\sigma = E'\sigma$, and is a *most general unifier* (*mgu*) if for every unifier θ of E and E', there exists substitution τ such that $\theta = \sigma\tau$. We denote the mgu of expressions E_1, E_2, E_1', E_2' with $\mathsf{mgu}(E_1, E_1')$ and $\mathsf{mgu}((E_1, E_2), (E_1', E_2'))$ for the mgu of tuples.

Saturation-Based Proving and Superposition. A saturation-based prover works with clauses. To prove a theorem G from axioms $A_1, \ldots, A_n$ (assumed to be clauses), the prover: (1) negates and clausifies G, obtaining clauses $\mathsf{cnf}(\neg G) = \{C_1, \ldots, C_m\}$; (2) forms the *saturation set* $S = \{C_1, \ldots, C_m, A_1, \ldots, A_n\}$; (3) repeats the following: chooses clauses $D_1, \ldots, D_k \in S$, uses an *inference rule* to derive a new clause D from $D_1, \ldots, D_k$, and adds D into S. Deriving the empty clause $\square$ at any point in step (3) concludes the proof, because it means

that $\neg G$ is inconsistent with the axioms, which is equivalent to G following from the axioms. A common *calculus* (a set of inference rules) used in saturation-based provers is the *superposition calculus* [20]. The calculus is parametrized by a *simplification order* $\succ$ on terms (see next paragraph) and a *selection function*, which selects in each non-empty clause some non-empty subset of literals. We denote selected literals by underlining them. An inference rule can be applied to the given premise(s) if the literals selected in the rule are also selected in the premise(s). For a certain class of selection functions, the superposition calculus 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).

Rewriting and Simplification Orders. A binary relation $\rightarrow$ over the set of terms is a *rewrite relation* if (i) $l \rightarrow r \Rightarrow l\theta \rightarrow r\theta$ and (ii) $l \rightarrow r \Rightarrow s[l] \rightarrow s[r]$ for any terms l, r, s and substitution θ. We write $\leftarrow$ to denote the inverse of $\rightarrow$. We call an ordered pair $l \rightarrow r$ a *rewrite rule* if (i) l is not a variable and (ii) l contains all variables that occur in r. A *rewrite system* R is a set of rewrite rules. We denote by $\rightarrow_R$ the smallest rewrite relation that contains R. A term l is *irreducible* in R if there is no r s.t. $l \rightarrow_R r \in R$. We denote with $\rightarrow_R^*$ the reflexive-transitive closure of $\rightarrow_R$. A term r is a *normal form* of a term l w.r.t R if $l \rightarrow_R^* r$ and r is irreducible in R. A *rewrite order* is a strict (irreflexive) rewrite relation. A *reduction order* is a well-founded rewrite order. We consider reduction orders which are total on ground terms; such orders are also called *simplification orders*. A *precedence relation*, denoted by $\gg$, is a total order on the signature Σ.

The *lexicographic path order (LPO)*, denoted by $\succ_{\mathsf{lpo}}$, is parameterized by a precedence relation $\gg$. Let s, t be terms with $s = f(s_1, ..., s_n)$ and $\succ_{\mathsf{lpo}}^{\mathsf{lex}}$ the standard lexicographic ordering extension of $\succ_{\mathsf{lpo}}$. We write $s \succ_{\mathsf{lpo}} t$ if:

1. t is a variable and a proper subterm of s, or
2. there is $i \in \{1, \ldots, n\}$ such that $s_i \succeq_{\mathsf{lpo}} t$, or
3. $t = f(t_1, ..., t_n)$, $(s_1, \ldots, s_n) \succ_{\mathsf{lpo}}^{\mathsf{lex}} (t_1, \ldots, t_n)$ and $s \succ_{\mathsf{lpo}} t_j$, for all $j \in \{1, \ldots, n\}$,or
4. $t = g(t_1, ..., t_m)$, $f \gg g$ and $s \succ_{\mathsf{lpo}} t_j$, for all $j \in \{1, \ldots, m\}$.

It is known that LPOs are simplification orders [20].

We view equalities as *bags*, finite multisets, and define and extend the term orderings on bags. The *bag extension* for an ordering $\succ$ on a set X is a binary relation on bags over X, denoted by $\succ^{\mathsf{bag}}$, defined as the smallest transitive relation on bags such that $\{x, y_1, \ldots, y_n\} \succ^{\mathsf{bag}} \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ if $x \succ x_i$ for all $i \in \{1, \ldots, m\}$ and $m \geq 0$. If $\succ$ is well-founded, then $\succ^{\mathsf{bag}}$ is too. We order equality literals by mapping each equality $s \approx t$ to the bag $\{s, t\}$ and each disequality $s \not\approx t$ to the bag $\{s, s, t, t\}$, and then using $\succ^{\mathsf{bag}}$ over these bags. Finally, we order clauses by taking the bag extension of the bag extension for literals. We only write $E_1 \succ E_2$ for two expressions E_1 and E_2 when it is clear from the context which ordering is used. We also write $E_1 \succeq E_2$ instead of $E_1 \succ E_2 \vee E_1 = E_2$.

3 The Synthesis Problem and How to Solve It

In this section, we adjust notions [15] to the purposes of our work. We assume a fixed signature Σ.

Definition 1 (Synthesis Specification). *Let Σ_c be a subset of Σ and φ a closed formula in first-order logic over Σ of the form:*

$$\forall \bar{x} \exists y. F[\bar{x}, y]. \tag{2}$$

We define a synthesis specification*, or simply* specification*, Λ as the pair $\langle \Sigma_c, \varphi \rangle$. We call any symbol in Σ_c* computable w.r.t. Λ*, or just* computable *when Λ is clear from the context. Further, any symbol in $\Sigma \setminus \Sigma_c$ is* uncomputable w.r.t. Λ*, or just* uncomputable*. We denote the set of uncomputable symbols by Σ_u.*

An expression containing an uncomputable symbol is called uncomputable*. Expressions that are not uncomputable are called* computable*.* ❑

To synthesize programs for specifications, we use an if-then-else term constructor, denoted by ite.

Definition 2 (ite and Program Terms). *We define the* conditional term constructor ite *as follows. Let F be a formula over Σ and p, q be terms. When F is true, then $\mathsf{ite}(F, p, q)$ is interpreted as the interpretation of p; otherwise, $\mathsf{ite}(F, p, q)$ is interpreted as the interpretation of q. We call F the* ite-condition*.*

We define program terms *as the smallest set $\mathcal{P}$ such that (i) $\mathcal{T} \subseteq \mathcal{P}$, and (ii) $\mathsf{ite}(l \approx r, p, q) \in \mathcal{P}$ for any $l, r \in \mathcal{T}$, and $p, q \in \mathcal{P}$. We denote program terms by p, q, possibly with indices. A term that does not contain a conditional term constructor is called a* simple term*.* ❑

We note that any term with the ite symbol containing arbitrary Boolean expressions as conditions can be translated into an equivalent program term in $\mathcal{P}$.

Definition 3 (Solution for a Specification). *Let $\Lambda = \langle \Sigma_c, \forall \bar{x} \exists y. F[\bar{x}, y]] \rangle$ be a specification. A computable $p[\bar{x}] \in \mathcal{P}$ is called a* solution *for the specification Λ if $\forall \bar{x}. F[\bar{x}, p[\bar{x}]\theta]$ is valid for an arbitrary substitution θ grounding $p[\bar{a}]$ for fresh constants $\bar{a}$.* ❑

Note that if $\bar{x}$ are the only variables in $p[\bar{x}]$, then the condition for $p[\bar{x}]$ being a solution reduces to $\forall \bar{x}. F[\bar{x}, p[\bar{x}]]$ being valid. On the other hand, if $p[\bar{x}]$ contains variables $\bar{z}$ other than $\bar{x}$, then $p[\bar{x}]$ represents a class of solutions, one for each grounding of $\bar{z}$.

Definition 4 (Realizable Specification). *Let Λ be a specification. We call Λ* realizable *if there exists a computable program term that is a solution to it.* ❑

We are now ready to state the problem that is the focus of the present paper.

The Synthesis Problem under Realizability Assumptions

Given a realizable specification Λ, the *synthesis problem under realizability assumptions* is the task of finding a solution to Λ.

In the following, we refer to the synthesis problem under realizability assumptions simply as *the synthesis problem.*

Saturation-based Synthesis Framework. One approach to solving synthesis problems is the saturation-based synthesis framework introduced in [15], which extends a saturation-based first-order theorem prover into a synthesizer. The framework constructs a program for specification Λ in parallel with searching for a proof that specification (2) holds. Intuitively, y in (2) represents the output for the inputs $\bar{x}$, and thus substitutions into y in the proof correspond to fragments of the sought program. To track these substitutions throughout the proof, the framework of [15] uses a mechanism called answer literals [7]. In this paper, we use constrained clauses, called *answer clauses*, rather than adding answer literals to clauses.

Definition 5 (Answer Clause). *Let C be a clause and p a program term. We call the expression $C[\![p]\!]$ an* answer clause. *We call p the* answer *for the answer clause $C[\![p]\!]$. We denote answer clauses with $\mathcal{C}$ and $\mathcal{D}$, possibly with indices. Given a substitution σ, we write $C[\![p]\!]\sigma$ to denote $C\sigma[\![p\sigma]\!]$.* ❏

W.l.o.g. we assume that axioms $A_1, \ldots, A_n$ are a part of specification (2); that is, $F[\bar{x}, y]$ is of the form $(A_1 \wedge \cdots \wedge A_n) \rightarrow G[\bar{x}, y]$. To derive a program, the framework first preprocesses (2) like a saturation-based prover would: the formula $\neg\forall\bar{x}\exists y.F[\bar{x}, y]$ is converted to the equivalent $\exists\bar{x}\forall y.\neg F[\bar{x}, y]$ and skolemized, obtaining the equisatisfiable $\forall y.\neg F[\bar{\alpha}, y]$,[1] which is then clausified, resulting into $\mathsf{cnf}(\neg F[\bar{\alpha}, y]) = \{C_1, \ldots, C_m\}$. The framework then extends the clauses $C_1, \ldots, C_m$, among which are also the axioms, into answer clauses $C_1[\![y]\!], \ldots, C_m[\![y]\!]$,[2] and initializes the saturation set $S = \{C_1[\![y]\!], \ldots, C_m[\![y]\!]\}$, called *the initial set from* Λ. We denote the consecutive steps of clausification and extending clauses into answer clauses by $\mathsf{cnf}(\neg F[\bar{\alpha}, y])[\![y]\!]$.

Definition 6 (Semantics of Answer Clauses). $C[\![p]\!]$ is true *with respect to a given specification $\Lambda = \langle \Sigma_c, \forall\bar{x}\exists y.F[\bar{x}, y]\rangle$, if the universal closure of $C \vee F[\bar{\alpha}, p]$ is true, where w.l.o.g. we assume $C[\![p]\!]$ and $F[\bar{\alpha}, y]$ to have disjoint sets of variables.*[3] ❏

To work with answer clauses, we use an extension of the superposition calculus [20]. Using this calculus, we *saturate* S like a saturation-based prover: we extend the set S by new clauses derived by rules from the calculus based on

[1] Since the skolems $\bar{\alpha}$ represent the input of the sought program, they are computable.

[2] In [15], only the clauses C containing y are extended into answer clauses $C[\![y]\!]$. In this paper we extend *all* clauses C in preprocessing, including axioms, into answer clauses $C[\![y]\!]$.

[3] [15] uses analogous semantics while denoting $C[\![p]\!]$ by $C \vee \mathsf{ans}(p)$.

premises already present in the set. We call S the *set saturated from* Λ. The work [15] proves that the saturation approach is sound: it only derives valid answer clauses, and thus when the answer clause $\square[\![p[\bar{\alpha}]]\!]$ is derived, it is guaranteed that $p[\bar{x}]$ is a solution for Λ. However, [15] makes no completeness claims. In the following section, we show an example for which the approach fails to derive a program even though a computable program exists.

4 Superposition with Synthesis

In this section, we present our new calculus SUPRA, adapted from [15]. We discuss the orderings and selection constraints that guide SUPRA (Sect. 4.1). We also introduce an abstraction mechanism that separates computable and uncomputable terms via an inference rule, in addition to SUPRA (Sect. 4.2).

4.1 The SUPRA Calculus

Our new calculus for Superposition with Realizability Assumptions, in short SUPRA, is summarized in Fig. 1. Compared to [15], we do not use so-called computable unifiers (abstracting unifiers preventing uncomputable symbols from appearing in the answer term), but most general unifiers and only allow an inference when the answer term after applying σ is computable. Further, we include the rule Sup_U, which unifies the answer terms p, q from premises, while [15] included a superposition rule which did not unify p, q but rather added a constraint $p \not\approx q$ into the resulting clause. Finally, for simplicity, we work only with equality predicates; thus, we do not include binary resolution or factoring rules as in [15].

We next establish soundness of SUPRA; this result guarantees that the programs derived by SUPRA are solutions of the synthesis specification.[4]

Theorem 1 (Soundness). *The SUPRA calculus is sound with respect to the semantics of answer clauses.*

It follows from Theorem 1 that SUPRA derives solutions to synthesis specifications.

Corollary 1 (Solution to the Synthesis Problem). *Given a synthesis specification* $\Lambda = \langle \Sigma_c, \forall \bar{x} \exists y. F[\bar{x}, y] \rangle$*, if SUPRA derives* $\square[\![p[\bar{\alpha}]]\!]$ *from the initial set of* Λ*, then* $p[\bar{x}]$ *is a solution for* Λ*.*

The rules of SUPRA are, however, not the only ingredient necessary to guarantee that a solution for a realizable specification will be derived. The following example illustrates how a derivation can get stuck.

[4] We provide detailed proofs for all our results in the extended version [11] of this paper.

$$(\mathsf{Sup}_C)\ \frac{\underline{l \approx r} \vee C[\![p]\!] \qquad \underline{s[l'] \mathrel{\dot{\approx}} t} \vee D[\![q]\!]}{s[r] \mathrel{\dot{\approx}} t \vee C \vee D[\![\mathsf{ite}(l \approx r, q, p)]\!]\sigma}\ \text{where}\ \begin{array}{l}(1)\ \sigma = \mathsf{mgu}(l, l')\\ (2)\ l' \text{ is not a variable}\\ (3)\ r\sigma \not\succeq l\sigma,\ t\sigma \not\succeq s[l']\sigma\\ (4)\ \mathsf{ite}(l \approx r, q, p)\sigma \text{ is computable}\end{array}$$

$$(\mathsf{Sup}_U)\ \frac{\underline{l \approx r} \vee C[\![p]\!] \qquad \underline{s[l'] \mathrel{\dot{\approx}} t} \vee D[\![q]\!]}{s[r] \mathrel{\dot{\approx}} t \vee C \vee D[\![p]\!]\sigma}\ \text{where}\ \begin{array}{l}(1)\ \sigma = \mathsf{mgu}((l, p), (l', q))\\ (2)\ l' \text{ is not a variable}\\ (3)\ r\sigma \not\succeq l\sigma,\ t\sigma \not\succeq s[l']\sigma\\ (4)\ p\sigma \text{ is a computable simple term}\end{array}$$

$$(\mathsf{EqRes})\ \frac{\underline{s \not\approx t} \vee C[\![p]\!]}{C[\![p]\!]\sigma}\ \text{where}\ \begin{array}{l}(1)\ \sigma = \mathsf{mgu}(s, t)\\ (2)\ p\sigma \text{ is computable}\end{array}$$

$$(\mathsf{EqFac})\ \frac{\underline{s \approx t} \vee l \approx r \vee C[\![p]\!]}{s \approx t \vee t \not\approx r \vee C[\![p]\!]\sigma}\ \text{where}\ \begin{array}{l}(1)\ \sigma = \mathsf{mgu}(s, l)\\ (2)\ t\sigma \not\succeq s\sigma,\ r\sigma \not\succ t\sigma\\ (3)\ p\sigma \text{ is computable}\end{array}$$

Fig. 1. Rules of SUPRA, where all maximal literals in a clause are selected (denoted underlined), and $C[\![p]\!]\sigma$ denotes the application of σ on $C[\![p]\!]$.

Example 1. Our motivating example corresponding to the specification (1) can be expressed as a synthesis specification with computable symbols f, s, p and v. That is, $\Lambda : \langle\{\mathsf{f}, \mathsf{s}, \mathsf{p}, \mathsf{v}\}, \varphi\rangle$. We consider predicate symbols as function symbols mapping values to $\{\top, \bot\}$ in a two-sorted logic. We will try to derive a solution for Λ using SUPRA. The initial set from Λ, where α is the Skolem constant introduced for x and numbers denote clause labels, is $\{(1)\ \alpha \approx \mathsf{f} \vee \underline{\alpha \approx \mathsf{s}}[\![y]\!], (2)\ \underline{\alpha \not\approx \mathsf{s}} \vee \mathsf{wp}[\![y]\!], (3)\ \underline{\alpha \not\approx \mathsf{f}} \vee \mathsf{wv}[\![y]\!], (4)\ \neg\underline{\mathsf{w}y}[\![y]\!]\}$. We use a selection function that selects either (i) all maximal literals or (ii) at least one negative literal.[5] Let $\succ$ be an LPO with the following symbol precedence: $\mathsf{s} \gg \mathsf{f} \gg \mathsf{p} \gg \mathsf{v} \gg \mathsf{w} \gg \alpha$. We obtain the following derivation, where we write SE for a Sup inference followed by an EqRes inference:

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha \approx \mathsf{f} \vee \underline{\alpha \approx \mathsf{s}}[\![y]\!]^{(1)} \qquad \underline{\alpha \not\approx \mathsf{s}} \vee \mathsf{wp}[\![y]\!]^{(2)}}{\alpha \approx \mathsf{f} \vee \mathsf{wp}[\![y]\!]}(\mathsf{SE}) \qquad \underline{\alpha \not\approx \mathsf{f}} \vee \mathsf{wv}[\![y]\!]^{(3)}}{\mathsf{wp} \vee \mathsf{wv}[\![y]\!]}(\mathsf{SE}) \qquad \neg\underline{\mathsf{w}y}[\![y]\!]^{(4)}}{\underline{\mathsf{wv}}[\![\mathsf{p}]\!]} \qquad \neg\underline{\mathsf{w}y}[\![y]\!]^{(4)}}{\text{(inference not possible)}}$$

Here we get stuck, as no more inferences are possible. Superposition into (4) $\neg\mathsf{w}y[\![y]\!]$ with $\mathsf{wv}[\![\mathsf{p}]\!]$ cannot be applied, because v and p do not unify; moreover, we cannot create an ite with condition wv, because $\mathsf{w} \in \Sigma_u$. However, as mentioned before, Λ is realizable, e.g. by the solution $\mathsf{ite}(x \approx \mathsf{f}, \mathsf{v}, \mathsf{p})$. We did not derive a solution due to the selection of computable literals in clauses which also contained uncomputable literals. ❏

[5] Such a selection function is called *well-behaved* and is sufficient to achieve refutational completeness in standard superposition reasoning.

Remark 1. Since [15] does not restrict selection nor simplification orders in any way, the failed derivation from Example 1 can be replicated by the calculus of [15]. Therefore, the calculus [15] does not always find a solution if one exists – it is not complete under realizability assumptions. ❏

Based on Example 1, we observe that we have to select uncomputable literals before computable ones. To this end, we make uncomputable terms bigger than computable terms in the simplification order and restrict the selection functions admissible for SUPRA to always select all maximal literals. We formally refine the simplification order used by SUPRA as follows.

Definition 7 (Partitioned Ordering). *An ordering $\succ$ on terms is called* partitioned *if for any ground uncomputable term s and any ground computable term t it holds that $s \succ t$.* ❏

We require that the simplification order that parameterizes SUPRA is partitioned. In the rest of the paper, we will use an LPO with a precedence function where any uncomputable symbol is greater than any computable symbol. Not only can these types of LPOs be used for SUPRA, but also specific KBOs that are partitioned orderings, see [11] for further discussion.

Example 2. Let $\succ$ be an LPO based on $\mathsf{w} \gg \mathsf{v} \gg \mathsf{p} \gg \mathsf{s} \gg \mathsf{f}$. With a selection function selecting all maximal literals w.r.t. $\succ$, we find a solution for Λ from Example 1:

$$\dfrac{\dfrac{\dfrac{\overset{(2)}{\alpha \not\approx \mathsf{s} \vee \underline{\mathsf{wp}}[\![y]\!]} \quad \overset{(4)}{\neg \underline{\mathsf{w}y}[\![y]\!]}}{\underline{\alpha \not\approx \mathsf{s}}[\![\mathsf{p}]\!]}\,(\mathsf{SE}) \quad \overset{(1)}{\alpha \approx \mathsf{f} \vee \underline{\alpha \approx \mathsf{s}}[\![y]\!]}}{\underline{\alpha \approx \mathsf{f}}[\![\mathsf{p}]\!]}\,(\mathsf{SE}) \quad \dfrac{\overset{(4)}{\neg \underline{\mathsf{w}y}[\![y]\!]} \quad \overset{(3)}{\alpha \not\approx \mathsf{f} \vee \underline{\mathsf{wv}}[\![y]\!]}}{\underline{\alpha \not\approx \mathsf{f}}[\![\mathsf{v}]\!]}\,(\mathsf{SE})}{\Box[\![\mathsf{ite}(\alpha \approx \mathsf{f}, \mathsf{v}, \mathsf{p})]\!]}\,(\mathsf{SE})$$

❏

4.2 Abstraction of Computable Terms in Uncomputable Literals

While our motivating example (1) shows that inferences with uncomputable literals should be applied before inferences with computable literals, sometimes this cannot be enforced, as some uncomputable equations can only be used in a superposition inference after performing a superposition into them with a computable equation.

Example 3. Consider constants a, b, c, d, e, unary symbols f, g, h, and the specification:

$$\langle \{a, b, c, d, e\},\ \exists y.(fd \not\approx e \vee (d \not\approx c \wedge gy \approx ga) \vee (fc \approx e \wedge hy \approx hb)) \rangle.$$

We perform the following derivation using an LPO $\succ$ with $h \gg g \gg f \gg e \gg d \gg c \gg b \gg a$. The leaves are obtained from the initial set of the above specification:

$$\dfrac{\dfrac{\overset{(1)}{\underline{fd \approx e[\![y]\!]}} \quad (\mathsf{ER})\ \dfrac{\overset{(2)}{d \approx c \vee \underline{gy \not\approx ga[\![y]\!]}}}{\underline{d \approx c[\![a]\!]}}}{(\mathsf{Sup}_U)\ \underline{fc \approx e[\![\mathsf{ite}\approx cya]\!]}} \quad (\mathsf{ER})\ \dfrac{\overset{(3)}{fc \not\approx e \vee \underline{hy \not\approx hb[\![y]\!]}}}{\underline{fc \not\approx e[\![b]\!]}}}{(\mathsf{Sup}_C)\ \text{(inference not possible)}}$$

The last step of the derivation is not possible, because the answers b and $\mathsf{ite}\approx cya$ do not unify, and we cannot construct an ite in the answer, because the condition $fc \approx e$ is uncomputable. ❏

The general observation we make from Example 3 is that we should resolve all uncomputable literals, even those containing computable subterms, before applying inferences with computable symbols. For this purpose, we *abstract the computable subterms of uncomputable literals*, thus separating reasoning with uncomputable and computable symbols. We do this by adding an auxiliary inference rule Abs, which replaces its premise by its consequence, indicated by the crossed-out premise. We *apply* Abs *exhaustively on each clause before we allow any further inferences*:

$$(\mathsf{Abs})\ \dfrac{\cancel{s[k] \mathrel{\dot{\approx}} t \vee C[\![p]\!]}}{s[x] \mathrel{\dot{\approx}} t \vee x \not\approx k \vee C[\![p]\!]}$$

where
(1) $s \not\preceq t$
(2) there is a substitution θ s.t. $s[k]\theta$ is uncomputable and $k\theta$ is computable
(3) no proper superterm of k in $s[k]$ satisfies property (2)
(4) k is not a variable
(5) x is a fresh variable

Lemma 1. *The rule* Abs *is sound with respect to the semantics of answer clauses.*

To apply the rule in practice, we use the following syntactic conditions in place of condition (2) of Abs.

Lemma 2. *Condition (2) of rule* Abs *holds iff (2a) k is computable, and (2b) either $s[k]$ is uncomputable, or it contains a variable not contained in k, and there exists at least one uncomputable term.*

Example 4. The derivation from Example 3 changes as follows when using Abs:

$$\dfrac{\dfrac{\dfrac{\overset{(2)}{\cancel{d \approx c \vee gy \not\approx ga[\![y]\!]}}}{(\mathsf{Abs})\ d \approx c \vee \underline{gy \not\approx gx} \vee x \not\approx a[\![y]\!]}}{(\mathsf{ER})\ \underline{d \approx c} \vee \underline{x \not\approx a}[\![x]\!]}}{(\mathsf{ER})\ \underline{d \approx c}[\![a]\!]} \quad \dfrac{\dfrac{\overset{(1)}{\cancel{fd \approx e[\![y]\!]}}}{(\mathsf{Abs})\ \underline{fx \approx e} \vee x \not\approx d[\![y]\!]}}{(\mathsf{SE})\ \underline{fx \approx e} \vee x \not\approx d[\![y]\!]} \quad \dfrac{\dfrac{\dfrac{\overset{(3)}{\cancel{fc \not\approx e \vee hy \not\approx hb[\![y]\!]}}}{(\mathsf{Abs})\ fc \not\approx e \vee \underline{hy \not\approx hx} \vee x \not\approx b[\![y]\!]}}{(\mathsf{ER})\ \underline{fc \not\approx e} \vee \underline{x \not\approx b}[\![x]\!]}}{(\mathsf{Abs})\ \underline{fz \not\approx e} \vee z \not\approx c \vee \underline{x \not\approx b}[\![x]\!]} \quad \dfrac{\dfrac{\dfrac{\overset{(3)}{\cancel{fc \not\approx e \vee hy \not\approx hb[\![y]\!]}}}{(\mathsf{Abs})\ fc \not\approx e \vee \underline{hy \not\approx hx} \vee x \not\approx b[\![y]\!]}}{(\mathsf{ER})\ \underline{fc \not\approx e} \vee \underline{x \not\approx b}[\![x]\!]}}{(\mathsf{Abs})\ \underline{fz \not\approx e} \vee z \not\approx c \vee \underline{x \not\approx b}[\![x]\!]}}{(\mathsf{ER})\ \underline{fz \not\approx e} \vee z \not\approx c[\![b]\!]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{(2)}{\cancel{d \approx c \vee gy \not\approx ga[\![y]\!]}}}{(\mathsf{Abs})\ d \approx c \vee \underline{gy \not\approx gx} \vee x \not\approx a[\![y]\!]}}{(\mathsf{ER})\ \underline{d \approx c} \vee \underline{x \not\approx a}[\![x]\!]}}{(\mathsf{ER})\ \underline{d \approx c}[\![a]\!]} \quad \dfrac{\dfrac{\overset{(1)}{\cancel{fd \approx e[\![y]\!]}}}{(\mathsf{Abs})\ \underline{fx \approx e} \vee x \not\approx d[\![y]\!]}}{(\mathsf{SE})\ \underline{fx \approx e} \vee x \not\approx d[\![y]\!]}}{(\mathsf{ER})\ \dfrac{y \not\approx d \vee y \not\approx c[\![b]\!]}{\underline{d \not\approx c}[\![b]\!]}}}{(\mathsf{SE})\ \square[\![\mathsf{ite}\approx cba]\!]}$$

Our new Abs rule allowed us to solve the specification of Example 3, deriving the program $\mathsf{ite}\approx cba$.

❏

5 Completeness of SUPRA Under Realizability Assumptions

In this section, we prove that the SUPRA calculus is *complete with respect to realizability assumptions*. In particular, we show that if there exists a solution to the specification Λ, then SUPRA used exhaustively with the rule Abs is guaranteed to derive some (possibly different) computable program, which is also a solution for Λ since SUPRA is sound (see Sect. 4.1). Our main result is the following theorem.

Theorem 2 (Completeness Under Realizability). *Let $\Lambda = \langle \Sigma_c, \forall \bar{x} \exists y. F[\bar{x}, y] \rangle$ be a specification. If Λ is realizable, then SUPRA with* Abs *derives an answer clause of the form $\square[\![p[\bar{\alpha}]]\!]$ from Λ where $p[\bar{x}]$ is a computable program term that is a solution to Λ.*

We prove Theorem 2 similarly to the completeness proof of the superposition calculus [21], by contradiction. We assume that S is a set saturated from Λ up to redundancy and abstracted (Definition 12), and that $\square[\![p[\bar{\alpha}]]\!] \notin S$ for any $p[\bar{\alpha}]$. Yet, we assume that Λ is realizable and that $t[\bar{x}]$ is a solution to it – i.e., that $\forall \bar{x}.F[\bar{x}, t[\bar{x}]]$ is valid. W.l.o.g. we assume $t[\bar{x}]$ does not contain any variables except for $\bar{x}$. We then consider a grounding of S using $t[\bar{\alpha}]$ (Definition 15) and construct an interpretation for it (Definition 10). However, since the grounding of S is equisatisfiable with $\neg F[\bar{\alpha}, t[\bar{\alpha}]]$, which is unsatisfiable because $\forall \bar{x}.F[\bar{x}, t[\bar{x}]]$ is valid, the interpretation should not be a model of S. We then consider the smallest clause $\mathbb{C}$ in the grounding that is false in the interpretation. Based on it, we either find an even smaller clause in the grounding that is also false in the interpretation, or show that the clause in S, instance of which is $\mathbb{C}$, should have been removed by Abs, since S is abstracted. In either case, we obtain a contradiction. We split this part of the proof into two steps: first, in Lemma 4 we show that all computable clauses in the grounding of S are satisfied in the model; then in Lemma 6 we extend this property to all uncomputable clauses.

Our construction involves substituting program terms into clauses – the term $t[\bar{\alpha}]$ is substituted for y in $\mathsf{cnf}(\neg F[\bar{\alpha}, y])[\![y]\!]$. If $t[\bar{\alpha}]$ contains ite terms, we have to unroll them by creating multiple instances of the clause and adding the condition(s) from the ite into them. We collect the unrolled condition literals in lists and separate them from the original clause, using so-called conditional clauses, as defined below.

Definition 8 (Conditional Clause, c-clause). *Let C be a clause and $\mathcal{L}$ a list of literals. A* conditional clause, c-clause *for short, is an expression of the form $C \triangledown \mathcal{L}$. We call $\mathcal{L}$ the* condition *of the c-clause $C \triangledown \mathcal{L}$. The c-clause $C \triangledown \mathcal{L}$ is logically equivalent to the clause $C \vee \bigvee_{L \in \mathcal{L}} L$. We denote c-clauses with $\mathbb{C}$ and $\mathbb{D}$. We use the notation $C\blacktriangledown$ to denote the c-clause $C \triangledown \epsilon$.* ❏

We extend the ordering $\succ$ over clauses to c-clauses as follows.

Definition 9 (Ordering on c-clauses). *We define an* ordering $\succ$ over c-clauses *as follows. First, we extend* $\succ$ *to lists of literals. We have* $\mathcal{L} \succ \epsilon$ *for any non-empty* $\mathcal{L}$ *and* $L\mathcal{L} \succ K\mathcal{K}$ *if either (i)* $L \succ K$ *or (ii)* $L = K$ *and* $\mathcal{L} \succ \mathcal{K}$. *We have* $C \triangledown \mathcal{L} \succ D \triangledown \mathcal{K}$ *if either (i)* $C \succ D$, *or (ii)* $C = D$ *and* $\mathcal{L} \succ \mathcal{K}$.[6] ❑

We now define a model for sets of c-clauses. As usual in superposition completeness proofs, we use rewrite systems as interpretations, where any equation $s \approx t$ is true in a rewrite interpretation R, denoted $R \vDash s \approx t$, if s and t have the same normal form in R. As usual, when s, t are ground, $R \vDash s \not\approx t \iff R \nvDash s \approx t$. A c-clause $C \triangledown \mathcal{L}$ is true in R if C or $\mathcal{L}$ contain at least one literal L such that $R \vDash L$.

Definition 10 (Model for c-clauses). *Let* S *be a set of ground c-clauses. For every c-clause* $\mathbb{C} \in S$, *we define in parallel two sets of term rewrite rules* $R_S^{\mathbb{C}}$ *and* $R_S^{\prec \mathbb{C}}$ *as* partial interpretations *by induction on the relation* $\succ$ *on ground c-clauses. First, we define:*

$$R_S^{\prec \mathbb{C}} := \bigcup_{\mathbb{D} \prec \mathbb{C}, \mathbb{D} \in S} R_S^{\mathbb{D}}.$$

A ground c-clause $\mathbb{C}$ *of the form* $\underline{l \approx r} \vee C \triangledown \mathcal{L}$ *is called* productive *if*

1. $\mathcal{L}$ *is computable,*
2. $l \approx r \vee C$ *is false in* $R_S^{\prec \mathbb{C}}$,
3. *if* $l \approx r \vee C$ *is not computable, then all literals in* $\mathcal{L}$ *are false in* $R_S^{\prec \mathbb{C}}$,
4. $l \approx r$ *is strictly maximal in* $\mathbb{C}$,
5. $l \succ r$,
6. C *is false in* $R_S^{\prec \mathbb{C}} \cup \{l \to r\}$,
7. l *is irreducible in* $R_S^{\prec \mathbb{C}}$.

In this case, we also say that $\mathbb{C}$ produces *the rule* $l \to r$. *Now we define*

$$R_S^{\mathbb{C}} := \begin{cases} R_S^{\prec \mathbb{C}} \cup \{l \to r\}, & \text{if } \mathbb{C} \text{ produces } l \to r; \\ R_S^{\prec \mathbb{C}}, & \text{otherwise.} \end{cases}$$

Finally, we define the total interpretation R_S for S *as* $\bigcup_{\mathbb{C} \in S} R_S^{\mathbb{C}}$. *As usual, an interpretation which satisfies a set of c-clauses* S *is called a model of* S. ❑

We state two standard properties about the model.

Lemma 3. *Let* S *be a set of ground c-clauses.*

1. R_S *is a convergent rewrite system.*
2. $R_S^{\mathbb{C}} \vDash \mathbb{C}$ *if and only if for all* $\mathbb{D} \succ \mathbb{C}$ *we have* $R_S^{\mathbb{D}} \vDash \mathbb{C}$, *if and only if* $R_S \vDash \mathbb{C}$.

We will construct a model for a set which is *saturated up to redundancy* and *abstracted*, with the following redundancy notions.

[6] This ordering preserves well-foundedness.

Definition 11 (Redundant Answer Clause/Inference). *An answer clause* $\mathcal{C}$ *is* redundant w.r.t. S *if every ground instance of* $\mathcal{C}$ *follows from smaller ground instances in* S*. Let* N *be the set of all ground instances of answer clauses in* S*. An inference* $\mathcal{C}_1, ..., \mathcal{C}_n \vdash \mathcal{D}$ *is* redundant w.r.t. S *if for each* θ *grounding for* $\mathcal{C}_1, \ldots, \mathcal{C}_n$ *and* $\mathcal{D}$ *either*

1. $\mathcal{D}\theta \succ \mathcal{C}_i\theta$ *for some* $1 \leq i \leq n$*, or*
2. $\mathcal{D}\theta$ *follows from the set* $\{C \mid C \in N$ *and* $\mathcal{C}_i\theta \succ C$ *for some* $1 \leq i \leq n\}$. ❑

Definition 12 (Saturation Up to Redundancy, Abstracted Set). *A set of answer clauses* S *is* saturated up to redundancy *if, given non-redundant answer clauses* $\mathcal{C}_1, ..., \mathcal{C}_n \in S$*, any SUPRA inference* $\mathcal{C}_1, ..., \mathcal{C}_n \vdash \mathcal{D}$ *is redundant w.r.t.* S*.*
If there is no clause $\mathcal{C} \in S$ *on which* Abs *would apply, then we call* S abstracted.❑

In the remainder of this section, we assume $\Lambda = \langle \Sigma_c, \forall \bar{x}.\exists y.F[\bar{x}, y] \rangle$ to be a fixed but arbitrary specification, and S a set of answer clauses saturated up to redundancy from Λ and abstracted. We assume $\square[\![p]\!] \notin S$ for any $p \in \mathcal{P}$. Note that the set of all ground instances of S might be unsatisfiable even if there is no computable solution for the specification, just an uncomputable one. Therefore, we cannot show refutational completeness of SUPRA. Instead, we show that from S, we can construct a counter-model for any computable program being a solution to Λ. Formally, we show that given any ground computable program term $t[\bar{\alpha}]$ (possibly using ite), the formula $\mathsf{cnf}(\neg F[\bar{\alpha}, t])$ is satisfiable. Towards this, we first eliminate ite from answer clauses $C_1[\![t]\!], \ldots, C_m[\![t]\!]$ coming from preprocessing, resulting in c-clauses.

Definition 13 (ite Normal Form). *Let* $\mathcal{C} = C[y][\![y]\!]$ *be an answer clause and* t *a program term. The* ite normal form *of* $\mathcal{C}$ *w.r.t.* t*, denoted* $\mathsf{iteNF}(\mathcal{C}, t)$*, is the set of c-clauses defined inductively as follows:*

1. *If* $C[y]$ *contains* y *and* t *is of the form* $\mathsf{ite}(L, s_1, s_2)$*, then it is*

$$\{D \bigtriangledown \hat{L}\mathcal{L} \mid D \bigtriangledown \mathcal{L} \in \mathsf{iteNF}(\mathcal{C}, s_1)\} \cup \{D \bigtriangledown L\mathcal{L} \mid D \bigtriangledown \mathcal{L} \in \mathsf{iteNF}(\mathcal{C}, s_2)\},$$

2. *otherwise, it is* $\{C[t]\blacktriangledown\}$. ❑

Intuitively, we obtain a c-clause $C[s] \bigtriangledown \mathcal{L}$ by elimination of ite from $C[\![t]\!]$ when $\mathcal{L}$ is the list of negations of ite-conditions needed to reach the branch term s in t.

For any ground computable program term t, we construct a set of ground c-clauses S' such that if S' is satisfiable, then $\mathsf{cnf}(\neg F[\bar{\alpha}, t])$ is also satisfiable. Then, we show that the total model R'_S satisfies S'. The computable part of S' is defined as follows.

Definition 14 (Computable Grounding). *We define the* computable grounding *of* S*, denoted* $\mathbf{cGr}(S)$*, as the set of c-clauses that contains:*

1. *all computable ground instances of c-clauses in* $\mathsf{iteNF}(\mathcal{C}, t')$ *for all* $\mathcal{C}$ *in the initial set of* Λ *and program terms* t'*, and*

2. *all computable ground c-clause instances of answer clauses in* S. ❑

We prove that the model $R_{\mathbf{cGr}(S)}$ satisfies all computable clauses in $\mathbf{cGr}(S)$. The proof is very similar to standard completeness proofs, since the rules of SUPRA for computable clauses correspond to the standard superposition rules: we do not apply any extra restrictions to inferences between computable clauses, and any inference between them results in a computable clause. By induction on $\succ$ over c-clauses, we obtain the following.

Lemma 4. *If* $\square[\![p]\!] \notin S$ *for any computable* $p \in \mathcal{P}$*, then* $R_{\mathbf{cGr}(S)} \vDash \mathbb{C}$ *for all* $\mathbb{C} \in \mathbf{cGr}(S)$.

To obtain S', we extend the set $\mathbf{cGr}(S)$ with uncomputable ground clauses. This set also depends on a computable program term t, and is called a grounding of S w.r.t. t.

Definition 15 (Grounding w.r.t. a Program Term). *Let* $t \in \mathcal{P}$ *be ground and* t' *be the normal form of* t *w.r.t.* $R_{\mathbf{cGr}(S)}$*. We define the* grounding of S w.r.t. program term t*, denoted* $\mathbf{Gr}(S,t)$*. In parallel, we define the* derivation length *for each uncomputable* $\mathbb{C}$ *in* $\mathbf{Gr}(S,t)$*, denoted* $\mathsf{dl}(\mathbb{C})$*.*
We define $\mathbf{Gr}(S,t)$ *as the minimal set of c-clauses containing* $\mathbf{cGr}(S)$ *such that:*

1 *For any answer clause* $\mathcal{C}$ *in the initial set of* Λ*, the set* $\mathbf{Gr}(S,t)$ *contains all ground instances of c-clauses in* $\mathsf{iteNF}(\mathcal{C}, t')$*.*
For any $\mathbb{C}$ *added to* $\mathbf{Gr}(S,t)$ *by this step, we define* $\mathsf{dl}(\mathbb{C}) = 0$*.*
2 *For any inference* $C_1[\![p_1]\!], \ldots, C_n[\![p_n]\!] \vdash C[\![p]\!]\sigma$ *such that* $C_i[\![p_i]\!] \in S$ *for all* $1 \leq i \leq n$*, and* $C[\![p]\!] \in S$*, if there is a substitution* θ *such that* $\sigma\theta = \theta$*,* $\mathbf{Gr}(S,t)$ *contains uncomputable c-clauses* $C_1\theta \bigtriangledown \mathcal{L}_1, \ldots, C_n\theta \bigtriangledown \mathcal{L}_n$*, and there is* j*,* $1 \leq j \leq n$*, such that for all* $1 \leq i \leq n$*, either* $\mathcal{L}_i = \epsilon$*, or* $\mathcal{L}_i = \mathcal{L}_j$*, then* $C\theta \bigtriangledown \mathcal{L}_j$ *is in* $\mathbf{Gr}(S,t)$*.*
Take the inference and θ *such that* $\max(\mathsf{dl}(C_1\theta \bigtriangledown \mathcal{L}_1), \ldots, \mathsf{dl}(C_n\theta \bigtriangledown \mathcal{L}_n))$ *is minimal. We define* $\mathsf{dl}(C\theta \bigtriangledown \mathcal{L}_j) = \max(\mathsf{dl}(C_1\theta \bigtriangledown \mathcal{L}_1), \ldots, \mathsf{dl}(C_n\theta \bigtriangledown \mathcal{L}_n)) + 1$. ❑

In the remainder of this section, we assume that t is an arbitrary but fixed ground computable program term. One key step in our main result is to show that inferences between uncomputable answer clauses with non-unifiable answers are not needed for a refutation that yields a computable answer. Towards this, we prove the following lemma about the grounding of S w.r.t. t, using induction on the derivation length of c-clauses.

Lemma 5. *There is a set of lists of computable ground literals* Δ *that satisfies the following properties:*

1. *Any uncomputable* $\mathbb{C} \in \mathbf{Gr}(S,t)$ *is of the form* $C \bigtriangledown \mathcal{L}$ *where* $\mathcal{L} \in \Delta \cup \{\epsilon\}$*.*
2. *For any* $\mathcal{L} \in \Delta \cup \{\epsilon\}$*, there is a ground computable simple term* $t_{\mathcal{L}}$ *such that for any uncomputable* $C' \bigtriangledown \mathcal{L} \in \mathbf{Gr}(S,t)$*, substitution* θ*, and clause* $C[\![p]\!] \in S$ *such that* $C\theta = C'$*, it holds that* p *is a simple term and* $p\theta = t_{\mathcal{L}}$ *or* p *is a variable not in* C*.*

3. *If* $\epsilon \notin \Delta$, *then for any uncomputable* $C'\blacktriangledown \in \mathbf{Gr}(S,t)$, *substitution* θ, *and clause* $C[\![p]\!] \in S$ *such that* $C\theta = C'$, *it holds that* p *is a variable not in* C.
4. *For any* $\mathcal{L}, \mathcal{K} \in \Delta$ *such that* $\mathcal{L} \neq \mathcal{K}$, $\mathcal{L}$ *and* $\mathcal{K}$ *contain a complementary literal.*

Intuitively, any list of literals $\mathcal{L}$ in Δ corresponds to the negation of ite-conditions needed to reach some branch term, i.e. a simple term s in t. The lemma states that:

(i) Any c-clause $\mathbb{C}$ in $\mathbf{Gr}(S,t)$ represents an instance of an answer clause $C[\![p]\!]$ in S with either the empty condition, meaning that p is any program term not depending on C, or with some conditions $\mathcal{L}$ in Δ such that s is an instance of p.
(ii) Any two lists of conditions in Δ contain a pair of complementary literals.

This ensures that inferences between uncomputable c-clauses corresponding to different branches from t are not needed, because if one of them is productive (Definition 10), the other one is necessarily satisfied by the model. Now we can prove that $R_{\mathbf{Gr}(S,t)}$, obtained by extending $R_{\mathbf{cGr}(S)}$ with uncomputable clauses, is a model of $\mathbf{Gr}(S,t)$, and thus also of $\neg F[\bar{\alpha}, t]$.

Lemma 6. *If* $\square[\![p[\bar{\alpha}]]\!] \notin S$ *for any computable program term* $p[\bar{\alpha}]$, *then the formula* $\neg F[\bar{\alpha}, t]$ *is satisfiable for any ground computable program term* t.

We conclude this section with the proof of our main result.

Proof (of Theorem 2). Since Λ is realizable, there is a program term $t[\bar{x}]$ with no variables except for $\bar{x}$, such that $\forall \bar{x}.F[\bar{x}, t[\bar{x}]]$ is valid. By contraposition of Lemma 6 it then follows that $\square[\![p[\bar{\alpha}]]\!] \in S$ for an abstracted set S saturated up to redundancy from Λ. Since SUPRA and Abs are sound (Theorem 1), $p[\bar{x}]$ is a solution to Λ. ❏

6 Related Work

Saturation-Based Synthesis. A saturation-based solution to the synthesis of recursion-free programs is introduced in [15]. We show that [15] is not complete. We introduce the SUPRA calculus to simplify the reasoning of [15], as (i) we assume all clauses to be answer clauses, and (ii) use most general unifiers instead of the arbitrary computable unifiers of [15] in inferences. Further, (iii) SUPRA implements a new abstraction rule Abs, needed for completeness under realizability assumptions.

Deductive Approaches to Synthesis. Our work, as well as [13,15], are based on the deductive synthesis approach of [19], adapted for resolution [17]. The mechanism in [28] restricts allowed programs, and subsumes our computability restrictions.

The work of [15] extends [13] with induction to support synthesis of recursive programs, with a proof of soundness and implementation, but without any

completeness guarantees. Our notion of realizability does not cover recursive programs, as their construction requires a mechanism for defining new functions not included in the computable symbols we work with.

A practical approach for synthesis in the presence of uncomputable symbols uses quantifier elimination and SMT solving [14], and requires $F[\bar{x}, y]$ to be quantifier-free in (2) but may use linear arithmetic. The approach is complete only when $F[\bar{x}, y]$ uses theories that admit quantifier elimination, which excludes e.g. equalities or uninterpreted functions.

The SyGuS [2] format allows specifying programs in a fragment of first-order logic extended by theories and optionally a grammar for the sought programs. Some prominent SyGuS solvers are cvc5 [24] and DryadSynth [16], as evidenced by the SyGuS competition [3]. Component-based synthesis methods construct programs from logical specifications using a predefined set of functions. The approach of [27] is deductive, while [9,29] use SMT solving. The GAPT framework [6] synthesizes programs from proofs, by computing witnesses of second-order formulas with quantifier elimination [1], or by extracting programs from natural deduction proofs in classical logic [25].

Related Completeness Results. Our completeness proof is inspired by standard completeness proofs for superposition [21]. Yet, our model construction only works for groundings that substitute computable program terms into the specification due to realizability assumptions. This relates our work to [28]. Finally, [30] proved that a general synthesis system for recursive programs cannot exist. This does not contradict our findings, as we claim completeness *under the assumption that a program exists.*

Restricted Inferences. A rule similar to Abs is used in [4,21] to disallow superpositions into certain terms, while [5] introduces unification constraints to the clause level that are similar to abstracted terms. In [23], the unification algorithm is extended by abstraction to enhance theory reasoning within superposition, abstracting terms on demand to enable inferences while postponing expensive theory reasoning.

7 Summary and Outlook

We tackle program synthesis and introduce the SUPRA calculus for saturation-based proof search. Our calculus revises [15] by using an abstract unification rule, which unravels computable subterms from uncomputable literals such that uncomputable literals can be resolved without disobeying the computability constraint of the calculus. We also use specific simplification orders and selection functions in SUPRA, ensuring that uncomputable literals are always selected if there are any in a clause. SUPRA changes the notion of clauses with answer literals to constrained clauses to make reasoning simpler. Not only do these adaptions retain soundness of SUPRA, but they enable proving completeness of SUPRA under the assumption that synthesis specifications have a solution. A

natural direction for future work is investigating completeness under realizability assumption for synthesis of recursive functions using calculi that make use of induction.

Acknowledgements. This research was funded in whole or in part by the ERC Consolidator Grant ARTIST 101002685, the Austrian Science Fund (FWF) 10.55776/DOC1345324, and the SBA Research COMET Center SBA-K1 NGC managed by the FFG.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Achammer, F., Hetzl, S., Schmidt, R.: Computing witnesses using the SCAN algorithm. In: CADE (2025). https://doi.org/10.1007/978-3-031-99984-0_27
2. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering. IOS Press (2015). https://doi.org/10.3233/978-1-61499-495-4-1
3. Alur, R., Fisman, D., Padhi, S., Reynolds, A., Singh, R., Udupa, A.: SyGuS-Comp (2019). `https://sygus.org/comp/2019/`
4. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. In: AAECC 5, 193–212 (1994). https://doi.org/10.1007/BF01190829
5. Bhayat, A., Schoisswohl, J., Rawson, M.: Superposition with delayed unification. In: CADE (2023). https://doi.org/10.1007/978-3-031-38499-8_2
6. Ebner, G., Hetzl, S., Reis, G., Riener, M., Wolfsteiner, S., Zivota, S.: System description: GAPT 2.0. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 293–301. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_20
7. Green, C.: Theorem-proving by resolution as a basis for question-answering systems. Mach. Intell. **4**, 183–205 (1969)
8. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL (2011). https://doi.org/10.1145/1926385.1926423
9. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011). https://doi.org/10.1145/1993498.1993506
10. Gulwani, S., Jojic, N.: Probabilistic inference of programs from input/output examples (2006). https://api.semanticscholar.org/CorpusID:59650668
11. Hajdu, M., Hozzová, P., Kovács, L., Wagner, E.M.: Completeness of synthesis under realizability assumptions using superposition (2026). https://arxiv.org/abs/2605.19683
12. Hozzová, P.: Integrating answer literals with AVATAR for program synthesis. In: Proceedings of the 7th and 8th Vampire Workshop (2024). https://doi.org/10.29007/vmn9
13. Hozzová, P., Amrollahi, D., Hajdu, M., Kovács, L., Voronkov, A., Wagner, E.M.: Synthesis of recursive programs in saturation. Automated Reas. (2024). https://doi.org/10.1007/978-3-031-63498-7_10
14. Hozzová, P., Bjørner, N.: Synthesiz3 This: an SMT-based approach for synthesis with Uncomputable symbols. In: FMCAD (2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_28

15. Hozzová, P., Kovács, L., Norman, C., Voronkov, A.: Program synthesis in saturation. In: CADE (2023). https://doi.org/10.1007/978-3-031-38499-8_18
16. Huang, K., Qiu, X., Shen, P., Wang, Y.: Reconciling enumerative and deductive program synthesis, p. PLDI (2020). https://doi.org/10.1145/3385412.3386027
17. Lee, R.C.T., Waldinger, R.J., Chang, C.L.: An improved program-synthesizing algorithm and its correctness. Commun. ACM (1974). https://doi.org/10.1145/360924.360967
18. Li, Y., Parsert, J., Polgreen, E.: Guiding enumerative program synthesis with large language models. In: CAV (2024). https://doi.org/10.1007/978-3-031-65630-9_15
19. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. TOPLAS (1980). https://doi.org/10.1145/357084.357090
20. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Handbook of Automated Reasonings. Elsevier and MIT Press (2001). https://doi.org/10.1016/B978-044450813-3/50009-6
21. Nieuwenhuis, R., Rubio, A.: Basic superposition is complete. In: Krieg-Brückner, B. (ed.) ESOP '92 (1992). https://doi.org/10.1007/3-540-55253-7_22
22. Reger, G.: Revisiting question answering in vampire. In: 4th Vampire Workshop (2018). https://doi.org/10.29007/fjc4
23. Reger, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: TACAS (2018).https://doi.org/10.1007/978-3-319-89960-2_1
24. Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C., Deters, M.: Refutation-based synthesis in SMT. Formal Methods Syst. Design **55**(2), 73–102 (2017). https://doi.org/10.1007/s10703-017-0270-2
25. Rietdijk, M.: Extracting programs from proofs using Friedman's A-translation and realizability (2018). https://repositum.tuwien.at/handle/20.500.12708/8452
26. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: CAV (2012). https://doi.org/10.1007/978-3-642-31424-7_44
27. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. In: CADE (1994). https://doi.org/10.1007/3-540-58156-1_24
28. Tammet, T.: Completeness of resolution for definite answers with case analysis. Comput. Sci. Logic (1995). https://doi.org/10.1007/BFb0022265
29. Tiwari, A., Gascón, A., Dutertre, B.: Program synthesis using dual interpretation. In: CADE (2015). https://doi.org/10.1007/978-3-319-21401-6_33
30. Voronkov, A.: On completeness of program synthesis systems. In: Börger, E., Jäger, G., Kleine Büning, H., Richter, M.M. (eds.) CSL 1991. LNCS, vol. 626, pp. 411–418. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0023785