Synthesis of Recursive Programs in Saturation

Petra Hozzová[1(✉)], Daneshvar Amrollahi[2], Márton Hajdu[1], Laura Kovács[1], Andrei Voronkov[1,3,4], and Eva Maria Wagner[1]

[1] TU Wien, Vienna, Austria
petra.hozzova@tuwien.ac.at
[2] Stanford University, Stanford, USA
[3] University of Manchester, Manchester, UK
[4] EasyChair, Manchester, UK

Abstract. We turn saturation-based theorem proving into an automated framework for recursive program synthesis. We introduce magic axioms as valid induction axioms and use them together with answer literals in saturation. We introduce new inference rules for induction in saturation and use answer literals to synthesize recursive functions from these proof steps. Our proof-of-concept implementation in the Vampire theorem prover constructs recursive functions over algebraic data types, while proving inductive properties over these types.

Keywords: Program Synthesis · Saturation · Superposition · Induction · Recursion · Theorem Proving

1 Introduction

Program synthesis is the task of constructing a program P satisfying a given specification F, ensuring that P is correct by design [20]. In this paper we work with a functional specification F of the input-output relation of a program P, *where F is given as a $\forall\exists$ formula in first-order logic* [1,20]. Validity of a specification formula F ensures that for every input value there exists an output value satisfying F, and therefore there is a function which for every input value gives such an output value. Our goal is to *automatically find a (possibly recursive) program that P* computes the output, while preserving F.

As a complementary approach to formal verification, synthesis is inherently more complex [28]. The complexity is further compounded when we consider reasoning about – and synthesizing – programs using recursion. As a remedy, in this paper we advocate for using automated first-order theorem proving as the reasoning back-end to (recursive) program synthesis.

The work [8] extended the saturation-based first-order theorem proving framework to *saturation-based synthesis framework*. The approach (i) uses saturation-based reasoning to prove that a specification F is valid; (ii) tracks the constructive parts of the proof of F; (iii) and uses them to synthesize a program P satisfying F. In this paper we complement [8] with support for *recursive program* synthesis. We use recent developments on automating induction in saturation [5,7,9,25] and construct recursive programs based on applications of induction.

C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 154–171, 2024.
https://doi.org/10.1007/978-3-031-63498-7_10

$$\begin{aligned}
\text{axioms: } & \mathsf{half}(0) \simeq 0 && \text{(H1)}\\
& \mathsf{half}(\mathsf{s}(0)) \simeq 0 && \text{(H2)}\\
& \forall x.\ \mathsf{half}(\mathsf{s}(\mathsf{s}(x))) \simeq \mathsf{s}(\mathsf{half}(x)) && \text{(H3)}\\
\text{specification: } & \forall x \exists y.\ \mathsf{half}(y) \simeq x && \text{(SD)}
\end{aligned}$$

Fig. 1. Axioms of half and the ∀∃-specification for the function computing double.

Illustrative Example. Consider the specification (SD) of Fig. 1, which describes the inverse of the half function over natural numbers. Given the axiomatization of half in Fig. 1, our approach synthesizes the recursive function double as a solution of (SD), defined as:

$$\begin{aligned}
\mathsf{double}(0) &\simeq 0\\
\forall x.\ \mathsf{double}(\mathsf{s}(x)) &\simeq \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))
\end{aligned} \tag{1}$$

The framework of [8] fails to synthesize a solution of (SD), as double is a recursive program. To the best of our knowledge, there exists no automated approach supporting recursive function synthesis from functional input-output specifications in full first-order logic.

This paper provides a solution in this respect by exploiting the constructive nature of induction. Intuitively, each case of an induction axiom tells us how to construct the desired program for the next recursive step using the program for the previous recursive step. We capture this construction recipe contained in the applications of induction in saturation-based proof search, by utilizing answer literals $\mathsf{ans}(r)$ [4]. When we use an induction axiom in the proof, we introduce a special term into the answer literal, serving for tracking the program corresponding to the induction axiom. As we prove the cases of the induction axiom, we capture their corresponding programs in the answer literal. Finally, when we derive a clause $C \vee \mathsf{ans}(r)$, where C only contains symbols allowed in a program, we convert the special tracker terms from r into recursive functions, and obtain a program for the initial specification conditioned on $\neg C$.

Contributions. We extend saturation-based first-order theorem proving with recursive program synthesis and bring the following contributions[1]:

- We introduce induction axioms, dubbed *magic axioms*, which capture the constructive nature of induction (Sect. 5).
- We convert the magic axioms into formulas used by a saturation-based framework to derive programs using recursion over algebraic data types, i.e., special cases of term algebras. We state necessary requirements for the calculus used in saturation and prove correctness of synthesized programs (Sect. 6).
- We present an extension of the superposition calculus that fulfills our necessary requirement and advocate for superposition reasoning for recursive function synthesis (Sect. 7).

[1] Proofs are given in the extended version [10] of our paper.

– We show that our approach, illustrated initially for natural numbers, naturally extends to programs over arbitrary term algebras (Sect. 8).
– We implement our work in the Vampire prover [16] and survey challenging examples it can synthesize (Sect. 9).

2 Preliminaries

We assume familiarity with standard multi-sorted first-order logic (FOL) with equality. We denote variables by x, y, z, w, u, terms by s, t, r, 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 $\Box$ for the *empty clause* which is logically equivalent to $\bot$. We write $\overline{L}$ for the literal complementary to L. By $\simeq$ we denote the equality predicate and write $t \not\simeq s$ as a shorthand for $\neg t \simeq s$. We include a conditional term constructor $\mathsf{if} - \mathsf{then} - \mathsf{else}$ in the language, as follows: given a formula F and terms s, t of the same sort, we write $\mathsf{if}\ F\ \mathsf{then}\ s\ \mathsf{else}\ t$ to denote the term s if F is true 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. Formulas with free variables are considered implicitly universally quantified, that is we consider closed formulas.

We use the standard semantics for FOL. For an interpretation function I, we denote the interpretation of a variable x, function symbol f and a predicate symbol p by x^I, f^I, p^I, respectively. We use the notation e^I, F^I also for the interpretation of expressions e and formulas F, respectively. Further, for a variable or a constant a and a value v, we denote by $I\{a \mapsto v\}$ the interpretation function I' such that $a^{I'} = v$ and $b^{I'} = b^I$ for any constant or variable $b \neq a$. 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.

We recall the standard notion of λ-expressions. Let t be a term and x a variable. Then $\lambda x.t$ denotes a λ-*expression*. For any interpretation I, we define $(\lambda x.t)^I$ as the function f given by $f(v) = t^{I\{x \mapsto v\}}$ for any value v. Moreover, we extend the notation of λ-expressions to also bind constants. Let c be a constant, then $\lambda c.t$ also denotes a λ-*expression*, and its interpretation $(\lambda c.t)^I$ is the function f given by $f(v) = t^{I\{c \mapsto v\}}$ for any value v.

A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two expressions E and E' if $E\theta = E'\theta$; θ is a *most general unifier* (*mgu*) if for every unifier η of E and E', there exists a substitution μ such that $\eta = \theta\mu$. We denote the mgu of E and E' with $\mathsf{mgu}(E, E')$.

We work with *term algebras* [27], in particular with the special classes of the algebraically defined data types of the natural numbers $\mathbb{N}$, lists $\mathbb{L}$, and binary trees $\mathbb{BT}$. [2] We denote the sorts of symbols and terms by : (colon), e.g., $f : \tau \rightarrow \alpha$ is a function symbol with domain τ and range α. To emphasize the sort τ of a quantified variable x, we write $\forall x \in \tau$ or $\exists x \in \tau$. For a term algebra sort τ, we denote its constructors with Σ_τ. We fix an arbitrary ordering on the constructors,

[2] Definitions of these term algebras are in the extended version [10] of this paper.

Superposition (Sup):

$$\frac{s \simeq t \vee C \quad L[s'] \vee D}{(L[t] \vee C \vee D)\theta}$$

where $\theta := \mathsf{mgu}(s, s')$.

Binary resolution (BR):

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

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

Factoring (F):

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

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

Equality resolution (ER):

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

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

Equality factoring (EF):

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

where $\theta := \mathsf{mgu}(s, s')$.

Fig. 2. Simplified superposition calculus $\mathbb{S}$up.

and denote the i-th constructor in the order by c_i, i.e., $\Sigma_\tau = \{c_1, \ldots, c_{|\Sigma_\tau|}\}$. For each c_i, we denote its arity with n_{c_i}. We denote with P_{c_i} the set of argument positions of c_i of the sort τ. We only consider the standard models of term algebras. Programs we synthesize may contain terminating recursive functions $f : \tau \to \alpha$, where τ is a term algebra type. We define such function f by providing a set of equalities $\{f(c(\overline{x})) \simeq t[\overline{x}, f(x_{j_1}), \ldots, f(x_{j_{|P_c|}})]\}_{c \in \Sigma_\tau}$, where $P_c = \{j_1, \ldots, j_{|P_c|}\}$, and t contains no occurrences of f except for the distinguished ones. An example of such a definition is (1).

Saturation and Superposition. Saturation-based proof search implements *proving by refutation* [16]: validity of F is proved by establishing unsatisfiability of $\neg F$. Saturation-based first-order theorem provers work with clauses, rather than with arbitrary formulas. To prove a formula F, the provers negate F and further skolemize it and convert it to clausal normal form (CNF). The CNF of $\neg F$ is denoted by $\mathsf{cnf}(\neg F)$, resulting in a set S of initial clauses. For example, the CNF of the negated and skolemized (SD) is

$$\mathsf{half}(y) \not\simeq \sigma, \tag{2}$$

where σ is a fresh constant used for skolemizing x, and y is implicitly universally quantified. Saturation provers *saturate* S by computing logical consequences of S with respect to a sound inference system $\mathcal{I}$. Whenever the empty clause $\square$ is derived, the set S of clauses is unsatisfiable and F is valid. We may extend the initial set S with additional clauses $C_1, \ldots, C_n$. If C is derived from this extended set, we say C is derived from S *under additional assumptions* $C_1, \ldots, C_n$.

The *superposition calculus* $\mathbb{S}$up [22] is the most common inference system for first-order logic with equality. Figure 2 shows a simplified version of $\mathbb{S}$up. The $\mathbb{S}$up 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).

3 Recent Developments in Saturation

In this section we summarize recent results relevant to our work.

Program Synthesis in Saturation. Synthesizing (non-recursive) programs in saturation has been initiated in [8]. Here, *computable* and *uncomputable* symbols in the signature are distinguished. Intuitively, computable symbols are those which are allowed to appear in a synthesized program. An expression is *computable* if all symbols it contains are computable. A symbol or an expression is *uncomputable* if it is not computable.

Let $A_1, \ldots, A_n$ be closed formulas. Then

$$A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x} \exists y. F[\overline{x}, y] \tag{3}$$

is a *(synthesis) specification with inputs* $\overline{x}$ *and output* y.

Consider a computable term $r[\overline{x}]$ such that $A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x}. F[\overline{x}, r[\overline{x}]]$ holds. Such an $r[\overline{x}]$ is called a *program* for (3) and a *witness for* y *in* (3). If $A_1 \wedge \ldots \wedge A_n \rightarrow \forall \overline{x}.(F_1 \wedge \ldots \wedge F_n \rightarrow F[\overline{x}, r[\overline{x}]])$ holds for computable formulas $F_1, \ldots, F_n$, then $\langle r[\overline{x}] \bigwedge_{i=1}^{n} F_i \rangle$ is a *program with conditions* $F_1, \ldots, F_n$ for (3).

Saturation-based theorem proving was extended in [8] to a *saturation-based program synthesis framework*. To this end, the clausified negated specification (3) is extended by an *answer literal* ans:

$$A_1 \wedge \ldots \wedge A_n \wedge \forall y.(\mathsf{cnf}(\neg F[\overline{\sigma}, y]) \vee \mathsf{ans}(y)) \tag{4}$$

The set of clauses (4) is then saturated. During saturation, upon deriving a clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$, where $r[\overline{\sigma}]$ is computable and $C[\overline{\sigma}]$ is computable and does not contain ans, the program $\langle r[\overline{x}] \neg C[\overline{x}] \rangle$ with conditions for (3) is recorded and the clause is replaced by $C[\overline{\sigma}]$. This step is called *answer literal removal* within saturation. Once saturation terminates by deriving the empty clause $\square$, the final program for (3) is constructed by composing the relevant recorded programs with conditions in a nested $\mathsf{if{-}then{-}else}$. To support derivation of such clauses $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ and to ensure that answer literals only have computable arguments, the work of [8] extended the superposition calculus $\mathbb{S}\mathrm{up}$ with new inference rules.

Induction in Saturation. Inductive reasoning has been integrated in saturation [5–7,9,25]. The main idea in this body of work is to apply induction by *theory lemma generation*: based on already derived formulas, generate a suitable induction axiom and add it to the search space. To this end, the following induction rule is used:

$$\frac{\overline{L}[t] \vee C}{F \rightarrow \forall x. L[x]}\ (\mathsf{Ind}),$$

where $L[t]$ is a ground literal, C is a clause, and $F \rightarrow \forall x. L[x]$ is a valid induction axiom. The conclusion of the Ind rule is clausified, yielding $\mathsf{cnf}(\neg F) \vee L[x]$. This

clause is resolved with the premise $\overline{L}[t] \vee C$ immediately after applying the Ind rule and the resulting clause $\mathsf{cnf}(\neg F) \vee C$ is added to the search space.

An example of a valid induction schema is the *structural induction axiom for natural numbers*, where $G[x]$ is any closed formula:

$$\big(G[\mathsf{0}] \wedge \forall y.(G[y] \rightarrow G[\mathsf{s}(y)])\big) \rightarrow \forall x.G[x] \tag{5}$$

When we instantiate the schema with $G[x] := L[x]$, we obtain an axiom that can be used in Ind. Since the rule requires $\overline{L}[t]$ to be ground, this instance of Ind cannot be applied on (2) and thus is not sufficient for proving (SD) of Fig. 1. To prove formulas with a free variable by induction, we extend Ind in Sect. 5.

Note that we can also use a complex formula $G[t]$ in place of the literal $L[t]$ in Ind, obtaining a more involved rule, possibly with multiple premises, similarly to a *mutli-clause induction rule* [7] or a *induction with arbitrary formulas* [6].

4 Saturation with Induction in Constructive Logic

We first summarize the key challenges our work resolves towards recursive synthesis in saturation, and then present our synthesis approach in Sects. 5–8.

The idea of extracting programs from proofs originates from results in constructive (intuitionistic) logic, starting with Kleene's realizability [14]. In constructive logic, provability of a formula $\forall \overline{x} \exists y.F[\overline{x}, y]$ implies that there is an algorithm which, given values for $\overline{x}$, outputs a value for y satisfying $F[\overline{x}, y]$.

We note that the structural induction axiom (5) over natural numbers has computational content, as follows. The program r for $\forall x.G[x]$ can be built from a program r_{0} for $G[\mathsf{0}]$ and a program r_{s} for $\forall y.(G[y] \rightarrow G[\mathsf{s}(y)])$ as:

$$r(\mathsf{0}) \simeq r_{\mathsf{0}}$$
$$r(\mathsf{s}(y)) \simeq r_{\mathsf{s}}(r(y))$$

For this to be useful, we need to first prove $G[\mathsf{0}]$, then prove $\forall x.(G[y] \rightarrow G[\mathsf{s}(y)])$, and then use the induction axiom to derive $\forall x.G[x]$. Such an approach towards constructing programs does not however work in saturation-based theorem proving, as saturation does not reduce goals to subgoals [2]. Rather, we add the induction axiom as a theory lemma to the proof search and continue saturation, so we do not have proofs of either $G[\mathsf{0}]$ or $\forall y.(G[y] \rightarrow G[\mathsf{s}(y)])$. Constructing programs during saturation becomes even more complex when using answer literals, because clauses generated during saturation may contain these literals. For example, if we try to extract a proof of $G[\mathsf{0}]$, we may find a proof with an answer literal in it.

To capture the constructive nature of induction and address the above challenges of program synthesis in saturation, we use the following trick. We modify the induction axiom so that it indirectly stores information about the programs for $G[\mathsf{0}]$ and $\forall y.(G[y] \rightarrow G[\mathsf{s}(y)])$. To do this, instead of adding the induction axiom (5), in Sect. 5 we add what we call a *magic axiom for* (5), where G has an additional argument for storing the program. In Sect. 6 we further convert our magic axioms into formulas to be used to derive recursive programs in saturation.

5 Induction with Magic Formulas

We first present our approach to *proving* formulas with a free variable by induction. We further extend this approach to *synthesis* in Sect. 6. While our approach works the same way with arbitrary term algebras, for the sake of clarity we first introduce our work for natural numbers and then for general term algebras in Sect. 8.

We use the following *magic axiom*:

$$\Big(\exists u_0.G[0,u_0] \wedge \forall y.\big(\exists w.G[y,w] \rightarrow \exists u_{\mathsf{s}}.G[\mathsf{s}(y),u_{\mathsf{s}}]\big)\Big) \rightarrow \forall z.\exists x.G[z,x] \qquad (6)$$

Note that all magic axioms are valid, as they are instances of the structural induction axiom (5) with the quantified formula $\exists x.G[t,x]$ in place of $G[t]$. The magicalness of (6) stems from its simple, yet powerful expressiveness: when used in proof search, the variables u_0, u_{s} in the antecedent capture the programs for the base and step cases, allowing us to construct a program for x in the consequent.

Using axiom (6), we introduce the following variant of the Ind rule:

$$\frac{\overline{L}[t,x] \vee C}{\Big(\exists u_0.L[0,u_0] \wedge \forall y.\big(\exists w.L[y,w] \rightarrow \exists u_{\mathsf{s}}.L[\mathsf{s}(y),u_{\mathsf{s}}]\big)\Big) \rightarrow \forall z.\exists x.L[z,x]}\ (\mathsf{MagInd})$$

where the only free variable of $L[t,x]$ is x and C does not contain x.

Example 1. Consider the specification (SD) from Fig. 1. To prove it using superposition, and not yet synthesize the function satisfying (SD) , we use the following magic axiom:

$$\Big(\exists u_0.\mathsf{half}(u_0) \simeq 0 \wedge \forall y.\big(\exists w.\mathsf{half}(w) \simeq y \rightarrow \exists u_{\mathsf{s}}.\mathsf{half}(u_{\mathsf{s}}) \simeq \mathsf{s}(y)\big)\Big) \rightarrow \forall z.\exists x.\mathsf{half}(x) \simeq z \quad (7)$$

To use (7) in saturation, we clausify it and skolemize the variables y, w, x as $\sigma_y, \sigma_w, \sigma_x(z)$, respectively. The following is a refutational proof of (SD) :

1. $\mathsf{half}(y) \not\simeq \sigma$ [negated and skolemized specification (SD)]
2. $\mathsf{half}(u_0) \not\simeq 0 \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{half}(\sigma_x(z)) \simeq z$ [MagInd with (7)]
3. $\mathsf{half}(u_0) \not\simeq 0 \vee \mathsf{half}(u_{\mathsf{s}}) \not\simeq \mathsf{s}(\sigma_y) \vee \mathsf{half}(\sigma_x(z)) \simeq z$ [MagInd with (7)]
4. $\mathsf{half}(u_0) \not\simeq 0 \vee \mathsf{half}(\sigma_w) \simeq \sigma_y$ [BR 1, 2]
5. $\mathsf{half}(u_0) \not\simeq 0 \vee \mathsf{half}(u_{\mathsf{s}}) \not\simeq \mathsf{s}(\sigma_y)$ [BR 1, 3]
6. $\mathsf{half}(u_0) \not\simeq 0 \vee \mathsf{half}(u_{\mathsf{s}}) \not\simeq \mathsf{s}(\mathsf{half}(\sigma_w))$ [Sup 4, 5]
7. $\mathsf{half}(u_0) \not\simeq 0 \vee \mathsf{half}(u_{\mathsf{s}}) \not\simeq \mathsf{half}(\mathsf{s}(\mathsf{s}(\sigma_w)))$ [Sup (H3), 6]
8. $\mathsf{half}(u_0) \not\simeq 0$ [ER 7]
9. $\square$ [BR 8, (H2)]

Hence, the magic axiom (6) is sufficient to prove (SD) . However, (6) does not suffice to synthesize the program for (SD) from the above proof. Similarly to [8], for synthesis we would use

$$\mathsf{half}(y) \not\simeq \sigma \vee \mathsf{ans}(y) \qquad (8)$$

instead of clause 1 and obtain a derivation similar to the one above, but with the answer literal $\mathsf{ans}(\sigma_x(\sigma))$. As σ_x is a fresh skolem function, it is uncomputable and not allowed in answer literals. Therefore, simply following the approach of [8] fails to synthesize a recursive program from the proof of (SD) . We address the challenge of program construction for the skolem function σ_x in Sect. 6. □

6 Programs with Primitive Recursion

We now construct recursive programs for proofs using induction over natural numbers (6). As mentioned in Sect. 4, the antecedent of the induction axiom gives us a recipe for constructing the program for the consequent. To capture this dependence of the consequent program x on the antecedent programs u_0, u_s, we convert the magic axiom (6) to its equivalent prenex normal form where $\forall u_0, u_\mathsf{s}$ precedes $\exists x$:

$$\exists y, w. \forall u_0, u_\mathsf{s}, z. \exists x. \Big(\big(G[0, u_0] \wedge (G[y, w] \rightarrow G[\mathsf{s}(y), u_\mathsf{s}]) \big) \rightarrow G[z, x] \Big) \tag{9}$$

The prenex form (9) of the magic axiom (6) allows us to record the dependency on the programs resulting from the base and step cases of induction. For that, we introduce a recursive operator to be used for constructing programs.

Definition 1 (Primitive Recursion Operator). Let $f_1 : \alpha$, and $f_2 : \mathbb{N} \times \alpha \rightarrow \alpha$. The *primitive recursion operator* R *for natural numbers and* α is:

$$\mathsf{R}(f_1, f_2)(0) \simeq f_1$$
$$\mathsf{R}(f_1, f_2)(\mathsf{s}(y)) \simeq f_2(y, \mathsf{R}(f_1, f_2)(y))$$

Lemma 2 (Recursive Witness). The expression $\mathsf{R}(u_0, \lambda y, w.u_\mathsf{s})(z)$ is a witness for the variable x in (9).

Lemma 2 ensures that we can construct a program for the consequent of the magic axiom given programs for the base case and the step case. We next integrate this construction into our synthesis framework using answer literals. For that we take a close look at the skolemization of induction axiom (9), and define skolem symbols, denoted via rec, for the variable x, capturing the recursive program.

Definition 3 (rec-Symbols). Consider formulas $G[t, x]$ with a single free variable $x : \alpha$ containing a term $t : \mathbb{N}$. For each such formula we introduce a distinct computable function symbol $\mathsf{rec}_{G[t,x]} : \alpha \times \alpha \times \mathbb{N} \rightarrow \alpha$. We will refer to such symbols $\mathsf{rec}_{G[t,x]}$ as rec-*symbols*. When the formula $G[t, x]$ is clear from the context or unimportant for the context, we will simply write rec instead of $\mathsf{rec}_{G[t,x]}$.

A term with a rec-symbol as the top-level functor is called a rec-*term*.

Definition 4 (Magic Formula). *The magic formula for* $G[t, x]$ is:

$$\forall u_0, u_\mathsf{s}, z. \Big(\big(G[0, u_0] \wedge (G[\sigma_y, \sigma_w] \rightarrow G[\mathsf{s}(\sigma_y), u_\mathsf{s}]) \big) \rightarrow G[z, \mathsf{rec}_{G[t,x]}(u_0, u_\mathsf{s}, z)] \Big) \tag{10}$$

It is easy to see that magic formula (10) is obtained by skolemizing the prenex normal form of magic axiom (9), where we replace the variables y, w by fresh constants σ_y, σ_w, and the variable x by a fresh $\mathsf{rec}_{G[t,x]}$-symbol. The constants σ_y, σ_w introduced in (10) are said to be *associated with the* $\mathsf{rec}_{G[t,x]}$*-term*. An occurrence of any skolem constant σ_y, σ_w is considered computable if it is an occurrence in the second argument of a $\mathsf{rec}_{G[t,x]}$-term which it is associated with.

We introduce additional requirements for reasoning with rec-terms to ensure that they always represent the recursive function to be synthesized.

Definition 5 (rec-Compliance). An inference system $\mathcal{I}$ is rec-*compliant* if:

1. $\mathcal{I}$ only introduces rec-terms in the instances of the magic formula (10),
2. $\mathcal{I}$ does not introduce uncomputable symbols into arguments of rec-terms in clauses it derives.

Using a rec-compliant inference system $\mathcal{I}$, we derive clauses containing rec-terms. These terms correspond to functions constructed using the operator R.

Definition 6 (Recursive Function Term). Let σ_y, σ_w be associated with $\mathsf{rec}(s_1, s_2, t)$. Then we call the term $\mathsf{R}(s_1, \lambda\sigma_y, \sigma_w.s_2)(t)$ the *recursive function term corresponding to* $\mathsf{rec}(s_1, s_2, t)$.

For a term r, we denote by r^{R} the expression obtained from r by iteratively replacing all rec-terms by their corresponding recursive function terms, starting from the innermost ones. Similarly, formula F^{R} denotes the formula F in which we replace all rec-terms by their corresponding recursive function terms.

Lemma 7 (Recursive Witness for Magic Formulas). Consider the formula obtained from (10) by replacing $\mathsf{rec}_{G[t,x]}(u_0, u_\mathsf{s}, z)$ by its corresponding recursive function term $\mathsf{R}(u_0, \lambda\sigma_y, \sigma_w.u_\mathsf{s})(z)$:

$$\forall u_0, u_\mathsf{s}, z. \\ \Big(\big(G[0, u_0] \wedge (G[\sigma_y, \sigma_w] \rightarrow G[\mathsf{s}(\sigma_y), u_\mathsf{s}])\big) \rightarrow G[z, \mathsf{R}(u_0, \lambda\sigma_y, \sigma_w.u_\mathsf{s})(z)]\Big) \tag{11}$$

For every interpretation I, there exists a mapping of skolem constants to values $\{\sigma_y \mapsto v_y, \sigma_w \mapsto v_w\}$ such that I extended by this mapping is a model of (11). As a consequence, formula (11) is satisfiable.

Lemma 7 implies that we can use formula (11) instead of (10) in derivation, while preserving the soundness of the derivations. Soundness of our approach to recursive program synthesis is given next.

Theorem 8 (Semantics of Clauses with Answer Literals and rec-terms). Let $C_1, \ldots, C_m$ be clauses and F a formula containing no answer literals and no rec-symbols. Let C be a clause containing no answer literals. Let $M_1, \ldots, M_l$ be magic formulas. Assume that using a sound rec-compliant inference system $\mathcal{I}$, we derive $C \vee \mathsf{ans}(r[\overline{\sigma}])$, where $r[\overline{\sigma}]$ is computable, from the set of clauses

$$\{\ C_1, \ldots, C_m,\ M_1, \ldots, M_l,\ \mathsf{cnf}(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y))\ \}.$$

Then

$$M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m \vdash C^{\mathsf{R}}, F[\overline{\sigma}, r^{\mathsf{R}}[\overline{\sigma}]].$$

That is, under the assumptions $M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m, \neg C^{\mathsf{R}}$, the computable expression $r^{\mathsf{R}}[\overline{x}]$ is a witness for y in $\forall \overline{x} \exists y.F[\overline{x}, y]$.

Based on Theorem 8, if the CNF of $A_1, \ldots, A_n$ is among $C_1, \ldots, C_m$, then $r^{\mathsf{R}}[\overline{x}]$ is a witness for y in (3) under the assumptions $M_1^{\mathsf{R}}, \ldots, M_l^{\mathsf{R}}, C_1, \ldots, C_m, \neg C^{\mathsf{R}}$. The following ensures that we can construct recursive programs with conditions.

Theorem 9 (Recursive Programs). Let $r[\overline{\sigma}]$ be a computable term, and $C[\overline{\sigma}], C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$ be ground computable clauses containing no answer literals and no rec-symbols. Assume that using a sound rec-compliant inference system $\mathcal{I}$, we derive the clause $C[\overline{\sigma}] \vee \mathsf{ans}(r[\overline{\sigma}])$ from the CNF of

$$\{\ A_1, \ldots, A_n,\ C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}],\ M_1, \ldots, M_l,\ \neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)\ \}$$

where $M_1, \ldots, M_l$ are magic formulas. Then,

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

is a program with conditions for (3).

From Theorem 9 we obtain the following key result on program synthesis.

Theorem 10 (Recursive Program Synthesis). Let $P_1[\overline{x}], \ldots, P_k[\overline{x}]$, where $P_i[\overline{x}] = \langle r_i^{\mathsf{R}}[\overline{x}] \bigwedge_{j=1}^{i-1} C_j[\overline{x}] \wedge \neg C_i[\overline{x}] \rangle$, be programs with conditions for (3), such that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable. Then the program $P[\overline{x}]$ defined as

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

is a program for (3).

7 Recursive Synthesis in Saturation

This section integrates the proving and synthesis steps of Sects. 5–6 into saturation. The crux of our approach is that instead of adding standard induction formulas to the search space, we add magic formulas.

Theorems 9–10 imply that to derive recursive programs, we can use any rec-compliant calculus, as long as the calculus supports derivation of clauses

$C \vee \mathsf{ans}(r)$, where r is computable and C is ground, computable, and contains no rec-terms nor answer literals. In our work we rely on the extended $\mathbb{S}$up calculus of [8], which we (i) further extend by adding magic formulas alongside standard induction formulas, (ii) make rec-compliant by disallowing inferences containing uncomputable rec-terms, and (iii) extend by adding more complex rules for introducing conditions into rec-terms[3]. We illustrate these steps by our running example.

Example 2. Using the extended $\mathbb{S}$up calculus, we synthesize the program for the specification of Fig. 1. With the magic formula corresponding to (7) ,

$$\forall u_0, u_\mathsf{s}, z.\\ \Big(\big(\mathsf{half}(u_0) \simeq \mathsf{0} \wedge (\mathsf{half}(\sigma_w) \simeq \sigma_y \rightarrow\ \mathsf{half}(u_\mathsf{s}) \simeq \mathsf{s}(\sigma_y))\big) \rightarrow \mathsf{half}(\mathsf{rec}(u_0, u_\mathsf{s}, z)) \simeq z\Big), \tag{12}$$

we obtain the following derivation[4]:

1. $\mathsf{half}(y) \not\simeq \sigma \vee \mathsf{ans}(y)$ [negated, skolemized specification with answer literal]
2. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{half}(\sigma_x(z)) \simeq z$ [MagInd with (12)]
3. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{half}(u_\mathsf{s}) \not\simeq \mathsf{s}(\sigma_y) \vee \mathsf{half}(\sigma_x(z)) \simeq z$ [MagInd with (12)]
4. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))$ [BR 1, 2]
5. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{half}(u_\mathsf{s}) \not\simeq \mathsf{s}(\sigma_y) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))$ [BR 1, 3]
6. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{half}(u_\mathsf{s}) \not\simeq \mathsf{s}(\mathsf{half}(\sigma_w)) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))$ [Sup 4, 5]
7. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{half}(u_\mathsf{s}) \not\simeq \mathsf{half}(\mathsf{s}(\mathsf{s}(\sigma_w))) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))$ [Sup (H3) , 6]
8. $\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \mathsf{ans}(\mathsf{rec}(u_0, \mathsf{s}(\mathsf{s}(\sigma_w)), \sigma))$ [ER 7]
9. $\mathsf{ans}(\mathsf{rec}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\sigma_w)), \sigma))$ [BR 8, (H2)]
10. $\square$ [answer literal removal 9]

The program recorded in step 10 of the proof is $\mathsf{rec}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\sigma_w)), x)^\mathsf{R} = \mathsf{R}(\mathsf{s}(\mathsf{0}), \lambda\sigma_w.\mathsf{s}(\mathsf{s}(\sigma_w)))(x) = f(x)$, where f is defined as:

$$f(\mathsf{0}) \simeq \mathsf{s}(\mathsf{0})$$
$$f(\mathsf{s}(n)) \simeq \mathsf{s}(\mathsf{s}(f(n)))$$

Note that while the synthesized program satisfies the specification (SD) , it does not match the expected definition of the double function from (1). Since the half function is rounding down, and the specification does not require the synthesized function to produce even results, the base case was resolved in step 9 with (H2) , leading to $f(\mathsf{0}) \simeq \mathsf{s}(\mathsf{0})$. As a result, we have $f(n) = \mathsf{s}(\mathsf{double}(n))$ for any n. $\square$

Example 2 demonstrates that specification (SD) has multiple solutions and saturation can find a solution different from the intended one. In the next example we modify the specification to have a single solution and synthesize it.

[3] The rules can be found in the extended version of this paper [10].
[4] For the fully detailed derivation, see [10].

Example 3. To synthesize the double function, we modify the specification:

$$\begin{aligned}
\text{additional axioms:}\quad & \mathsf{even}(\mathsf{0}) && \text{(E1)}\\
& \neg\mathsf{even}(\mathsf{s}(\mathsf{0})) && \text{(E2)}\\
& \forall x.\ (\mathsf{even}(\mathsf{s}(\mathsf{s}(x))) \leftrightarrow \mathsf{even}(x)) && \text{(E3)}\\
\text{new specification:}\quad & \forall x \exists y.\ (\mathsf{half}(y) \simeq x \wedge \mathsf{even}(y)) && \text{(SD')}
\end{aligned}$$

After negating and skolemizing (SD') and adding the answer literal, we obtain:

$$\mathsf{half}(y) \not\simeq \sigma \vee \neg\mathsf{even}(y) \vee \mathsf{ans}(y) \tag{13}$$

In this case we use the magic axiom for the conjunction $G[t, x] := \mathsf{half}(x) \simeq t \wedge \mathsf{even}(x)$:

$$\begin{aligned}
&\Big(\exists u_0.(\mathsf{half}(u_0) \simeq \mathsf{0} \wedge \mathsf{even}(u_0)) \wedge\\
&\quad \forall y.\big(\exists w.(\mathsf{half}(w) \simeq y \wedge \mathsf{even}(w)) \rightarrow \exists u_\mathsf{s}.(\mathsf{half}(u_\mathsf{s}) \simeq \mathsf{s}(y) \wedge \mathsf{even}(u_\mathsf{s}))\big)\Big)\\
&\rightarrow \forall z.\exists x.(\mathsf{half}(x) \simeq z \wedge \mathsf{even}(x))
\end{aligned} \tag{14}$$

We clausify the magic formula corresponding to (14), and further resolve it with the premise (13) to obtain:

$$\begin{gathered}
\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \neg\mathsf{even}(u_0) \vee \mathsf{half}(\sigma_w) \simeq \sigma_y \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))\\
\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \neg\mathsf{even}(u_0) \vee \mathsf{even}(\sigma_w) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))\\
\mathsf{half}(u_0) \not\simeq \mathsf{0} \vee \neg\mathsf{even}(u_0) \vee \mathsf{half}(u_\mathsf{s}) \not\simeq \mathsf{s}(\sigma_y) \vee \neg\mathsf{even}(u_\mathsf{s}) \vee \mathsf{ans}(\mathsf{rec}(u_0, u_\mathsf{s}, \sigma))
\end{gathered}$$

The refutation of these clauses follows a similar course to the proof in Example 2. However, u_0 occurring in the literal $\neg\mathsf{even}(u_0)$ forces the proof to use (H1) instead of (H2) , and thus the final derived answer literal will be $\mathsf{rec}(\mathsf{0}, \mathsf{s}(\mathsf{s}(\sigma_w)), \sigma)$, corresponding exactly to the function definition of double from (1). Note that a derivation of this program in this case requires a saturation prover to apply induction on conjunctions of literals. □

8 Generalization to Arbitrary Term Algebras

Our approach from Sects. 5– 7 generalizes naturally to arbitrary term algebras. This section summarizes the key parts of this generalization. [5]

Let τ be a (possibly polymorphic) term algebra with constructors $\{c_1, \ldots, c_n\}$, where we denote the sort of each c_i by $\tau_{i,1} \times \cdots \times \tau_{i,n_{c_i}} \rightarrow \tau$,

[5] We state all definitions, lemmas and theorems in the appendix of the extended version of our paper [10].

and $P_{c_i} = \{j_1, \ldots, j_{|P_{c_i}|}\}$ for each $i = 1, \ldots, n$. Let α be any sort. The *magic axiom for* $G[t, x]$, where $t : \tau, x : \alpha$, is:

$$\Big(\bigwedge_{c \in \Sigma_\tau} \forall_{i=1}^{n_c} y_{c,i}.((\bigwedge_{j \in P_c} \exists w_{c,j}.G[y_{c,j}, w_{c,j}]) \to \exists u_c.G[c(\overline{y_c}), u_c]) \Big) \to \forall z.\exists x.G[z, x] \quad (15)$$

The corresponding *magic formula* uses the skolem function $\mathsf{rec}_{G[t,x]} : \alpha^{n_c} \times \tau \to \alpha$:

$$\forall_{c \in \Sigma_\tau} u_c.\forall z.\Big(\bigwedge_{c \in \Sigma_\tau} (\bigwedge_{j \in P_c} G[\sigma_{y_{c,j}}, \sigma_{w_{c,j}}] \to G[c(\overline{\sigma_{y_c}}), u_c]) \to G[z, \mathsf{rec}_{G[t,x]}(\overline{u}, z)] \Big) \quad (16)$$

Note that each $\sigma_{y_{c_i,j}}, \sigma_{w_{c_i,j}}$ introduced in (16) is considered computable only in the ith argument of its associated rec-term. We define the *recursion operator* R *for* τ *and* α analogously to Definition 1:

$$\begin{aligned}
\mathsf{R}(f_1, \ldots, f_n)(c_1(\overline{x})) &\simeq f_1(x_1, \ldots, x_{n_{c_1}}, \mathsf{R}(f_1, \ldots, f_n)(x_{j_1}), \ldots, \mathsf{R}(f_1, \ldots, f_n)(x_{j_{|P_{c_1}|}})) \\
&\ldots \\
\mathsf{R}(f_1, \ldots, f_n)(c_n(\overline{x})) &\simeq f_n(x_1, \ldots, x_{n_{c_n}}, \mathsf{R}(f_1, \ldots, f_n)(x_{j_1}), \ldots, \mathsf{R}(f_1, \ldots, f_n)(x_{j_{|P_{c_n|}}}))
\end{aligned}$$

where for each i we have $f_i : \tau_{i,1} \times \cdots \times \tau_{i,n_{c_i}} \times \alpha^{|P_{c_i}|} \to \alpha$. Using R, we state an analogue of Lemma 7:

Lemma 11 (Recursive Witness for Magic Formulas Using τ). Consider the formula obtained from (16) by replacing $\mathsf{rec}_{G[t,x]}(\overline{u}, z)$ by its corresponding recursive function term:

$$\begin{aligned}
&\forall_{c \in \Sigma_\tau} u_c.\forall z.\Big(\bigwedge_{c \in \Sigma_\tau} (\bigwedge_{j \in P_c} G[\sigma_{y_{c,j}}, \sigma_{w_{c,j}}] \to G[c(\overline{\sigma_{y_c}}), u_c]) \\
&\to G[z, \mathsf{R}(\lambda_{i=1}^{n_{c_1}} \sigma_{y_{c_1,i}}.\lambda_{k \in P_{c_1}} \sigma_{w_{c_1,k}}.u_{c_1}, \ \ldots, \ \lambda_{i=1}^{n_{c_n}} \sigma_{y_{c_n,i}}.\lambda_{k \in P_{c_n}} \sigma_{w_{c_n,k}}.u_{c_n})(z)] \Big)
\end{aligned} \quad (17)$$

For every interpretation, there exists its extension by some $\{\sigma_{y_{c,i}} \mapsto v_{y,c,i}, \sigma_{w_{c,k}} \mapsto v_{w,c,k}\}_{c \in \Sigma_\tau, i \in \{1,\ldots,n_c\}, k \in P_c}$ such that the extension is a model of (17). As a consequence, formula (17) is satisfiable.

Using Lemma 11, we derive the analogues of Theorems 8–10 for an arbitrary term algebra τ. We then employ magic formulas (16) in MagInd when in the premise $L[t, x] \vee C \vee \mathsf{ans}(r[x])$ we have $t : \tau$. We finally note that our synthesis method generalizes also to sorts other than term algebras, as long as the induction axiom used for the sort carries the constructive meaning described in Sect. 4.

9 Implementation and Examples

Implementation. We extended the first-order theorem prover Vampire [16] with a proof-of-concept implementation of our method for recursive program

synthesis in saturation. Our implementation consists of approximately 1,100 lines of C++ code and is available online at https://github.com/vprover/vampire/tree/synthesis-recursive.

We implemented the MagInd rule as well as a version of MagInd using a magic axiom with base case $\mathsf{s}(0)$ for natural numbers and $\mathsf{cons}(a, \mathsf{nil})$ for any a for lists. To support synthesis requiring induction on specifications $\neg F[t, x]$, where $F[t, x]$ is an arbitrary formula with the only free variable x, we use an encoding as follows. We change the specification $\forall \overline{x} \exists y. F[\overline{x}, y]$ to $\forall \overline{x} \exists y. p(\overline{x}, y)$, where p is a fresh uncomputable predicate, and we add an axiom $\forall \overline{x}, y. (p(\overline{x}, y) \leftrightarrow F[\overline{x}, y])$.

Table 1. Synthesis examples using natural numbers $\mathbb{N}$, lists $\mathbb{L}$ and binary trees $\mathbb{BT}$. The x-variables in the program and synthesized definitions are the inputs. While our framework synthesizes all these examples, our implementation in Vampire only synthesizes those marked with "✓". Note that for "Length of 2 concatenated lists" we consider $+\!\!+$ to be uncomputable.

Specification	Program	Synthesized definitions	Vampire
Double: $\forall x \in \mathbb{N}. \exists y \in \mathbb{N}.$ $(\mathsf{half}(y) \simeq x \wedge \mathsf{even}(y))$	$f(x)$	$f(0) \simeq 0$ $f(\mathsf{s}(n)) \simeq \mathsf{s}(\mathsf{s}(f(n)))$	✓
Associativity of addition: $\forall x_1, x_2, x_3 \in \mathbb{N}. \exists y \in \mathbb{N}.$ $(x_1 + x_2) + x_3 \simeq x_1 + y$	$f(x_3)$	$f(0) \simeq x_2$ $f(\mathsf{s}(n)) \simeq \mathsf{s}(f(n))$	✓
Subtraction with condition: $\forall x_1, x_2 \in \mathbb{N}. \exists y \in \mathbb{N}.$ $(x_2 < x_1 \rightarrow x_2 + y \simeq x_1)$	$f(x_2)$	$f(0) \simeq x_1$ $f(\mathsf{s}(n)) \simeq \mathsf{p}(f(n))$	✓
Floored square root: $\forall x \in \mathbb{N}. \exists y \in \mathbb{N}.$ $(y \cdot y \leq x \wedge x < \mathsf{s}(y) \cdot \mathsf{s}(y))$	$f(x)$	$f(0) \simeq 0$ $f(\mathsf{s}(n)) \simeq \mathsf{if}\ \mathsf{s}(n) \simeq \mathsf{s}(f(n)) \cdot \mathsf{s}(f(n))$ $\mathsf{then}\ \mathsf{s}(f(n))\ \mathsf{else}\ f(n)$	✗
Floored division: $\forall x_1, x_2 \in \mathbb{N}. \exists y \in \mathbb{N}. (x_2 \not\simeq 0 \rightarrow$ $(y \cdot x_2 \leq x_1 \wedge x_1 < \mathsf{s}(y) \cdot x_2))$	$f(x_1)$	$f(0) \simeq 0$ $f(\mathsf{s}(n)) \simeq \mathsf{if}\ \mathsf{s}(n) \simeq \mathsf{s}(f(n)) \cdot x_2$ $\mathsf{then}\ \mathsf{s}(f(n))\ \mathsf{else}\ f(n)$	✗
Length of 2 concatenated lists: $\forall x_1, x_2 \in \mathbb{L}. \exists y \in \mathbb{N}.$ $y \simeq \mathsf{len}(x_1 x_2)$	$f(x_1)$	$f(\mathsf{nil}) \simeq \mathsf{len}(x_2)$ $f(\mathsf{cons}(n, l)) \simeq \mathsf{s}(f(l))$	✓
Last element of a list: $\forall x \in \mathbb{L}. \exists y \in \mathbb{N}. (x \not\simeq \mathsf{nil} \rightarrow$ $\exists z \in \mathbb{L}. x \simeq z\mathsf{cons}(y, \mathsf{nil}))$	$f(x)$	$f(\mathsf{cons}(n, \mathsf{nil})) \simeq n$ $l \not\simeq \mathsf{nil} \rightarrow f(\mathsf{cons}(n, l)) \simeq f(l)$	✓
Prefix of a list given its suffix: $\forall x_1, x_2 \in \mathbb{L}. \exists y \in \mathbb{L}.$ $(\mathsf{suff}(x_2, x_1) \rightarrow x_1 \simeq y x_2)$	$f(x_2)$	$f(\mathsf{nil}) \simeq x_1$ $f(\mathsf{cons}(n, l)) \simeq g(f(l))$ $g(\mathsf{cons}(n, \mathsf{nil})) \simeq \mathsf{nil}$ $l \not\simeq \mathsf{nil} \rightarrow g(\mathsf{cons}(n, l)) \simeq \mathsf{cons}(n, g(l))$	✗
Maximum element of a list: $\forall x \in \mathbb{L}. \exists y \in \mathbb{N}. (x \not\simeq \mathsf{nil} \rightarrow$ $(\mathsf{in}(y, x) \wedge \forall k \in \mathbb{N}. (\mathsf{in}(k, x) \rightarrow k \leq y))$	$f(x)$	$f(\mathsf{cons}(n, \mathsf{nil})) \simeq n$ $l \not\simeq \mathsf{nil} \rightarrow f(\mathsf{cons}(n, l)) \simeq \mathsf{if}\ f(l) < n$ $\mathsf{then}\ n\ \mathsf{else}\ f(l)$	✗
Maximum element of a tree: $\forall x \in \mathbb{BT}. \exists y \in \mathbb{N}.$ $(\mathsf{in}(y, x) \wedge \forall k \in \mathbb{N}. (\mathsf{in}(k, x) \rightarrow k \leq y)$	$f(x)$	$f(\mathsf{leaf}(n)) \simeq n$ $f(\mathsf{bt}(l, n, r)) \simeq$ $\mathsf{if}\ f(l) < f(r)\ \mathsf{then}$ $\mathsf{if}\ f(l) < n\ \mathsf{then}$ $\mathsf{if}\ f(r) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(r)$ $\mathsf{else}\ f(r)$ $\mathsf{else\ if}\ f(r) < n\ \mathsf{then}$ $\mathsf{if}\ f(l) < n\ \mathsf{then}\ n\ \mathsf{else}\ f(l)$ $\mathsf{else}\ f(l)$	✗

Examples. Our implementation can synthesize the programs for the specifications (SD) and (SD'). We also synthesize further examples over the term algebras[6] of natural numbers $\mathbb{N}$, lists $\mathbb{L}$, and binary trees $\mathbb{BT}$. We display the specifications alongside the programs synthesized by our framework in Table 1. Our framework synthesizes programs for each of the examples[7] , yet our implementation supports so far only a limited set of magic formulas; therefore, the "Vampire" column of Table 1 lists which examples are solved in practice.

Experimental Comparison. To the best of our knowledge, no other approach in program synthesis supports the setting we consider: functional relational specifications of recursive programs, given in full first-order logic, without user-defined templates. For this reason, we could not compare the practical aspects of our work with other techniques, but overview related works in Sect. 10. In particular, we note that the tools surveyed in overview in Sect. 10 support a more restrictive/decidable logic than the the full first-order setting exploited in our approach. As such, the benchmarks of Table 1 cannot be translated into the input languages of techniques surveyed in Sect. 10.

10 Related Work

Our approach is conceptually different from existing methods in recursive program synthesis, as we are not restricted to decidable logical fragments, nor to user-defined program templates. Our work supports program specifications in full first-order logic (with theories) and does not require syntactic templates for the programs to be synthesized. In the sequel, we only discuss related approaches that support *full automation* in program synthesis, without templates or user guidance.

We extend the recursion-free synthesis framework of [8], while exploiting ideas from deductive synthesis [17,20,29] using answer literals [4]. We bring recursive program synthesis into the landscape of saturation-based proving and construct programs from saturation proofs with magic axioms. Unlike our setting, the works of [19,29] construct recursive programs from proofs by induction, by reducing the program specification to subgoals corresponding to the cases of the induction axiom. Modern first-order theorem provers mostly implement saturation-based proof search, which however does not support a goal-subgoal architecture. Our approach integrates induction directly into saturation and enables automated reasoning with term algebras.

[6] See the extended version of this paper [10] for term algebra constructors and signatures, and for axiomatization and lemmas for the used predicates and functions.

[7] We provide the full derivations of the synthesized programs in [10].

Fully automated methods supporting recursive program synthesis include Synquid [23], Leon [15], Jennisys [18], SuSLik [24], Cypress [12], Burst [21], and Syntrec [11]. Except for Burst and Syntrec, all these works decompose goals into subgoals. Our work complements these methods, by turning saturation into a recursive synthesis framework over first-order theories. As such, our work also differs from Synquid, where term enumeration combined with type checking is used over program specifications within decidable logics. Leon uses recursive schemas corresponding to our recursive operator R, instantiates them by candidate program terms, and checks if they satisfy the specification. Unlike Leon, we support a complete handling of quantifiers via superposition reasoning. Jennisys uses a verifier to generate input-output examples, which differs from our setting of using inductive formulas as logical specifications. Burst generates programs by composition from existing ones, using quantifier-free fragments of first-order logic. Contrarily to this, we support full first-order logic and induction, without using subgoal proof strategies. Finally, we note that Syntrec guarantees bounded/relative correctness of the synthesized programs (using syntactic program templates), while our approach proves correctness of the synthesized program without further restrictions.

The syntax-guided synthesis (SyGuS) framework [1] supports specifications for recursive functions and can encode our examples from Sect. 9. However, to the best of our knowledge, SyGuS methods, including the SMT-based approach of [26], do not support recursive synthesis. While the semantics-guided synthesis framework [13] also supports recursive functions, its (to the best of our knowledge) only solvers Messy [13] and Messy-Enum [3] synthesize programs from input-output examples and using grammars, respectively, rather than purely from logical specifications.

11 Conclusions

We extend saturation-based framework to recursive program synthesis by utilizing the constructive nature of induction axioms. We introduce magic axioms as a tracking mechanism and seamlessly integrate these axioms into saturation. We then construct correct recursive programs using answer literals in saturation, as also demonstrated by our proof-of-concept implementation. Extending our work with tailored handling of (more general) magic axioms, and respective superposition inferences, is an interesting line for future work. Devising and implementing further, and potentially more general, synthesis rules and induction schemes is another task for future research, allowing us to further strengthen the practical use of our work.

Acknowledgements. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien SecInt Doctoral College, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, and the Amazon Research Award 2023 QuAT.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, pp. 1–25 (2015)
2. Bonacina, M.P.: A Taxonomy of theorem-proving strategies. In: Artificial Intelligence Today: Recent Trends and Developments, pp. 43–84 (1999). https://doi.org/10.1007/3-540-48317-9_3
3. D'Antoni, L., Hu, Q., Kim, J., Reps, T.: Programmable program synthesis. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 84–109. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_4
4. Green, C.: Theorem-proving by resolution as a basis for question-answering systems. Mach. Intell. **4**, 183–205 (1969)
5. Hajdu, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with Generalization in Superposition Reasoning. In: CICM, pp. 123–137 (2020) https://doi.org/10.1007/978-3-030-53518-6_8
6. Hajdu, M., Kovács, L., Rawson, M., Voronkov, A.: The Vampire Approach to Induction. In: Practical Aspects of Automated Reasoning (2022)
7. Hajdu, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: FMCAD, pp. 1–10 (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_34
8. Hozzová, P., Kovács, L., Norman, C., Voronkov, A.: Program synthesis in saturation. In: CADE, pp. 307–324 (2023)
9. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: CADE, pp. 361–377 (2021)
10. Hozzová, P., Amrollahi, D., Hajdu, M., Kovács, L., Voronkov, A., Wagner, E.M.: Synthesis of Recursive Programs in Saturation. EasyChair Preprint no. 12145, EasyChair (2024)
11. Inala, J.P., Polikarpova, N., Qiu, X., Lerner, B.S., Solar-Lezama, A.: Synthesis of recursive ADT transformations from reusable templates. In: TACAS, pp. 247–263 (2017). https://doi.org/10.1007/978-3-662-54577-5_14
12. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: PLDI, pp. 944–959 (2021). https://doi.org/10.1145/3453483.3454087
13. Kim, J., Hu, Q., D'Antoni, L., Reps, T.: Semantics-guided synthesis. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434311
14. Kleene, S.: On the interpretation of intuitionistic number theory. J. Symb. Log. **10**, 109–124 (1945)
15. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA, pp. 407–426 (2013). https://doi.org/10.1145/2509136.2509555
16. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV, pp. 1–35 (2013)
17. Lee, R.C.T., Waldinger, R.J., Chang, C.L.: An improved program-synthesizing algorithm and its correctness. Commun. ACM **4**, 211–217 (1974). https://doi.org/10.1145/360924.360967
18. Leino, K.R.M., Milicevic, A.: Program extrapolation with Jennisys. In: OOPSLA, pp. 411–430. OOPSLA '12 (2012). https://doi.org/10.1145/2384616.2384646
19. Manna, Z., Waldinger, R.: Fundamentals of deductive program synthesis. IEEE Trans. Softw. Eng. **18**(8), 674–704 (1992). https://doi.org/10.1109/32.153379
20. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980). https://doi.org/10.1145/357084.357090

21. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up Synthesis of Recursive Functional Programs using Angelic Execution. **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498682
22. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Handbook of Automated Reasoning, vol. I, pp. 371–443. Elsevier and MIT Press (2001)
23. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. ACM SIGPLAN Notices **51**(6), 522–538 (2016). https://doi.org/10.1145/2980983.2908093
24. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. **3**(POPL), 1–30 (2019). https://doi.org/10.1145/3290385
25. Reger, G., Voronkov, A.: Induction in Saturation-Based Proof Search. In: CADE, pp. 477–494 (2019)
26. Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C.W., Deters, M.: Refutation-based synthesis in SMT. Formal Methods Syst. Des. **55**(2), 73–102 (2019). https://doi.org/10.1007/S10703-017-0270-2
27. Rybina, T., Voronkov, A.: A decision procedure for term algebras with queues. ACM Trans. Comput. Log. **2**(2), 155–181 (2001). https://doi.org/10.1145/371316.371494
28. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010). https://doi.org/10.1145/1706299.1706337
29. Tammet, T.: Completeness of resolution for definite answers. J. Logic Comput. **5**(4), 449–471 (1995)