EasyChair Preprint

№ 8182

# Towards Synthesis in Superposition

Petra Hozzová, Laura Kovács and Andrei Voronkov



June 4, 2022

# Towards Synthesis in Superposition

Petra Hozzová
petra.hozzova@tuwien.ac.at
TU Wien

Laura Kovács
laura.kovacs@tuwien.ac.at
TU Wien

Andrei Voronkov
andrei@voronkov.com
University of Manchester and EasyChair

## ABSTRACT

We present our ongoing developments in synthesizing recursion-free programs using the superposition reasoning framework in first-order theorem proving. Given a first-order formula as a program specification, we use a superposition-based theorem prover to establish the validity of this formula and, along this process, synthesize a function that meets the specification. To this end, we modify the rules of the superposition calculus to synthesize program fragments corresponding to individual clauses derived during the proof search. If a proof is found, we extract a program based on the found (correctness) proof. We implemented our approach in the first-order theorem prover Vampire and successfully evaluated it on a few examples.

## KEYWORDS

Program Synthesis, Superposition Reasoning, Automated Deduction

## 1 INTRODUCTION

Superposition-based theorem provers were recently extended with inductive reasoning for both inductively defined datatypes [2, 4, 5, 8, 15] and integers [6]. We believe that automating inductive reasoning in the superposition framework opens up new directions for using first-order reasoners in program analysis and synthesis. This extended abstract is a first step in this direction: we demonstrate how a first-order theorem prover can automatically synthesize recursion-free programs.

Program synthesis is the task of producing a *computable* program given a formal specification of the input-output relation that the program should satisfy. It was noted already many years ago that forall-exists first-order formulas $\forall\overline{x}.\exists y.F[\overline{x}, y]$ can be used as such specifications [3, 11], expressing that "for all inputs $\overline{x}$ there exists an output $y$ such that the relation $F[\overline{x}, y]$ between the inputs and the output holds". More recent approaches to synthesis include using templates and a correctness oracle computing counterexamples for each incorrect candidate program [13, 16]. In this paper we focus on the former method. The main idea of our approach is to utilize the superposition reasoning framework to prove the given specification and simultaneously construct a program that satisfies the specification.

The technique we use is based on answer literals introduced in [3] and further developed by adding `if–then–else` constructs in [17]. Compared to [17], we extend this method in several ways to include theories and equality reasoning and making distinction between computable and non-computable symbols, since only the former can be used in programs. We implement it in the superposition-based first-order theorem prover Vampire [10] based on the previous implementation of question answering in Vampire [14].

Given the page limit, we will sometimes trade space for mathematical rigour, omit details and not define some notions that we use and discuss (such as program or computable expressions). Nonetheless, all we present here can be made completely formal.

## 2 SOME THEORY

Suppose we have a specification of the form

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

where $F$ is a first-order formula. We know that, if this formula is valid in some theory, in every model of this theory there is a function $f$ such that for all elements $\overline{\sigma}$ of the model, $f(\overline{\sigma})$ holds. When the theory has the standard model (e.g., arithmetic) and (some) functions and predicates are computable in tis model, we are interested in finding a computer program $f$ with this property. If we have a language for expressing programs, then it suffices to find an expression $r[\overline{x}]$ in this language such that

$$\forall\overline{x}.F[\overline{x}, r[\overline{x}]]$$

is valid in the theory. This idea comes from constructive logic and Kleene's realizability [7], and any expression $r$ with this property is called a *realizer* of (1). Realizers can be extracted from constructive proofs but using constructive proofs for program synthesis turned out to be impractical.

Instead of search for constructive proofs, we will use ordinary (classical) proofs, but during proof search will make sure that derived formulas are guaranteed to have realizers. A further bit of magic is added by using the superposition calculus in a creative way, where instead of searching for a proof we simply saturate input clauses until we find a clause of a special form, in a way similar to [9].

Let us add a new uninterpreted unary predicate symbol $\mathsf{ans}$ (*answer predicate*) and new (skolem) constants $\overline{\sigma}$. Suppose we can prove the formula

$$\forall y.(\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)) \implies \mathsf{ans}(t[\overline{\sigma}]),$$

where $t[\overline{\sigma}]$ is a ground term denoting a program. Since $\mathsf{ans}$ is uninterpreted, we can replace $\mathsf{ans}(y)$ by any formula. Let us replace it by $y \neq t[\overline{\sigma}]$. We obtain

$$\forall y.(\neg F[\overline{\sigma}, y] \vee y \neq t[\overline{\sigma}]) \implies t[\overline{\sigma}] \neq t[\overline{\sigma}]),$$

hence $\forall y.(\neg F[\overline{\sigma}, y] \vee y \neq t[\overline{\sigma}])$ is unsatisfiable, and so $\exists y.(F[\overline{\sigma}, y] \wedge y = t[\overline{\sigma}])$ is valid. This formula is equivalent to $F[\overline{\sigma}, t[\overline{\sigma}]]$. Since $\overline{\sigma}$ are fresh symbols, we obtain that the formula $\forall\overline{x}.F[\overline{x}, t[\overline{x}]]$ is valid, and hence $t[\overline{x}]$ is a realizer.

The good news is that one can use the superposition calculus saturation algorithm to implement this idea. To search for a proof of (1), we saturate the set of clauses obtained from $\neg F[\overline{\sigma}, y]$ with the aim of obtaining an empty clause (Section 3). To search for a realizer, we saturate the set of clauses obtained from $\neg F[\overline{\sigma}, y] \vee \mathsf{ans}(y)$ with

**Superposition:**

$$\frac{l = r \vee C \quad L[l'] \vee D}{(L[r] \vee C \vee D)\theta} \qquad \frac{l = r \vee C \quad s[l'] \neq t \vee D}{(s[r] \neq t \vee C \vee D)\theta} \qquad \frac{l = r \vee C \quad s[l'] = t \vee D}{(s[r] = t \vee C \vee D)\theta}$$

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

**Binary resolution:**

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

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

**Factoring:**

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

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

**Equality resolution:**

$$\frac{s \neq t \vee C}{C\theta}$$

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

**Equality factoring:**

$$\frac{s = t \vee s' = t' \vee C}{(s = t \vee t \neq t' \vee C)\theta}$$

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

**Figure 1: The superposition calculus $\mathbb{S}$up for first-order logic with equality.**

the aim of obtaining a clause of the form $\mathtt{ans}(t[\overline{\sigma}])$, where $t$ is a term denoting a program (Section 4).

One can also use a tuple of output terms $\overline{y}$ instead of a single variable $y$. In this case we use $\mathtt{ans}(\overline{y})$ instead of $\mathtt{ans}(y)$.

## 3 SUPERPOSITION-BASED PROOF SEARCH

We assume familiarity with *standard multi-sorted first-order logic with equality*. Variables are denoted with $x$, $y$, skolem constants with $\sigma$, terms with $t, s$, all possibly with indices. We assume a distinguished *integer sort*, denoted by $\mathbb{Z}$. When we use standard integer predicates $<, \leq, >, \geq$, we assume that they denote the corresponding interpreted integer predicates with their standard interpretations. Additionally we assume a conditional term constructor: If $A$ is an atom and $s, t$ are terms of the same sort, then $\mathtt{if}\ A\ \mathtt{then}\ s\ \mathtt{else}\ t$ is a term of the same sort interpreted in the standard way – as the interpretation of $s$ if the interpretation of $A$ is *true*, and the interpretation of $t$ otherwise.[1]

A *literal* is an atom or its negation. A disjunction of literals is a *clause*. We denote atoms by $A$, literals by $L$, clauses by $C, D$ and reserve the symbol $\Box$ for the *empty clause* which is logically equivalent to $\bot$. We denote the *clausal normal form* of a formula $F$ by $\mathtt{cnf}(F)$.

A *substitution* $\theta$ is a mapping from variables to terms. A substitution $\theta$ is a *unifier* of two terms $s$ and $t$ if $s\theta = t\theta$, and is a *most general unifier* (*mgu*) if for every unifier $\eta$ of $s$ and $t$, there exists substitution $\mu$ s.t. $\eta = \theta\mu$. We denote the mgu of $s$ and $t$ with $\mathtt{mgu}(s, t)$. We write $F[x_1, \ldots, x_n]$ to denote that the formula $F$ contains $k_1, \ldots, k_n$ occurrences of the variables $x_1, \ldots, x_n$, respectively, with $k_i \geq 0$ for all $i$. For simplicity, $F[t_1, \ldots, t_n]$ denotes the formula $F[x_1, \ldots, x_n]\theta$, where $\theta$ maps each $x_i$ to the term $t_i$.

### 3.1 Saturation and Superposition

We briefly introduce saturation-based proof search [10].

First-order theorem provers work with clauses. Given a set $S$ of input clauses, first-order provers *saturate* $S$ by computing all logical consequences of $S$ with respect to a sound inference system $\mathcal{I}$. The saturated set of $S$ is called the *closure* of $S$ and the process of computing the closure of $S$ is called *saturation*. If the closure of $S$ contains the empty clause $\Box$, the original set $S$ of clauses is unsatisfiable. Note that a saturation algorithm proves validity of $B$ by establishing unsatisfiability of $\neg B$; we refer to this proving process as a *refutation* of $\neg B$.

The *superposition calculus*, denoted as $\mathbb{S}$up, is the most common inference system employed by saturation-based first-order theorem provers for first-order logic with equality [12], such as the theorem prover Vampire [10]. A summary of superposition inference rules is given in Figure 1. Due to a lack of space, we present simplified rules of the calculus. The superposition calculus $\mathbb{S}$up is *sound*. It is also *refutationally complete* for the logic of uninterpreted functions: for any unsatisfiable set of clauses $S$, the empty clause can be derived from $S$.

### 3.2 Question Answering Using Answer Literals

Answer literals have been used in Vampire in the past for question answering [14]. Given a question in the form $\exists\overline{y}.F[\overline{y}]$, Vampire saturates $\forall\overline{y}.\mathtt{cnf}(\neg F[\overline{y}] \vee \mathtt{ans}(\overline{y}))$. While the the answer literals are in general treated as any other literal, they are never selected to actively participate in any inferences – they are only carried over and substituted into (as parts of $C$ or $D$ in the inference rules from Figure 1). However, if Vampire derives a clause only containing answer literals $\mathtt{ans}(\overline{t_1}) \vee \cdots \vee \mathtt{ans}(\overline{t_m})$, it means that that the question is provable, and it extracts the (possibly disjunctive) answer $\overline{t_1} \vee \cdots \vee \overline{t_m}$. This answer means that $F[\overline{t_1}] \vee \cdots \vee F[\overline{t_m}]$ holds. In particular, if $n = 1$, then $\overline{t_1}$ is a definite answer, that is, $F[\overline{t_1}]$ holds.

## 4 PROGRAM SYNTHESIS IN THE SUPERPOSITION CALCULUS

The programs that we aim to synthesize from specification (1) correspond to terms not containing any variables except for $\overline{x}$ (the input variables) and possibly using $\mathtt{if-then-else}$.

Compared to Vampire's previous use of answer literals, we need to exclude disjunctive answers, that is, clauses having more than one answer literal. Such clauses may appear by applying rules with more than premise to clauses with a single answer literal. To solve this problem, we modify inference rules with two premises containing answer literals as follows. Let $\mathtt{ans}(\overline{r_1})$ and $\mathtt{ans}(\overline{r_2})$ be answer literals of the first and the second premise, respectively. We replace the binary resolution rule from Figure 1 to *binary resolution with answer literals*, abbreviated AnsBR:

$$\frac{A \vee C \vee \mathtt{ans}(\overline{r_1}) \quad \neg A' \vee D \vee \mathtt{ans}(\overline{r_2})}{(C \vee D \vee \mathtt{ans}(\mathtt{if}\ A\ \mathtt{then}\ \overline{r_2}\ \mathtt{else}\ \overline{r_1}))\theta}\ \text{(AnsBR)}$$

That is, instead of creating a clause with two answer literals we combine the realizers represented by $\overline{r_1}, \overline{r_2}$ into a single realizer

[1] By abusing the notation, we use $\mathtt{if}\ A\ \mathtt{then}\ \overline{s}\ \mathtt{else}\ \overline{t}$ to denote the tuple $(\mathtt{if}\ A\ \mathtt{then}\ s_1\ \mathtt{else}\ t_1, \ldots, \mathtt{if}\ A\ \mathtt{then}\ s_n\ \mathtt{else}\ t_n)$.

$(a)$ $y < \sigma_1 \vee y < \sigma_2 \vee y \neq \sigma_1 \vee \mathsf{ans}(y)$ [input]

$(b)$ $y < \sigma_1 \vee y < \sigma_2 \vee y \neq \sigma_2 \vee \mathsf{ans}(y)$ [input]

$(c)$ $\neg x < x$ [< axiom]

$(d)$ $\neg x_1 < x_2 \vee \neg x_0 < x_1 \vee x_0 < x_2$ [< axiom]

$(e)$ $\sigma_1 < \sigma_1 \vee \sigma_1 < \sigma_2 \vee \mathsf{ans}(\sigma_1)$ [ER (a)]

$(f)$ $\sigma_2 < \sigma_1 \vee \sigma_2 < \sigma_2 \vee \mathsf{ans}(\sigma_2)$ [ER (b)]

$(g)$ $\sigma_1 < \sigma_2 \vee \mathsf{ans}(\sigma_1)$ [BR (c), (e)]

$(h)$ $\sigma_2 < \sigma_1 \vee \mathsf{ans}(\sigma_2)$ [BR (c), (f)]

$(i)$ $\neg x_0 < \sigma_2 \vee x_0 < \sigma_1 \vee \mathsf{ans}(\sigma_2)$ [BR (d), (h)]

$(j)$ $\sigma_1 < \sigma_1 \vee \mathsf{ans}(\mathtt{if}\ \sigma_1 < \sigma_2\ \mathtt{then}\ \sigma_2\ \mathtt{else}\ \sigma_1)$ [AnsBR (g), (i)]

$(k)$ $\mathsf{ans}(\mathtt{if}\ \sigma_1 < \sigma_2\ \mathtt{then}\ \sigma_2\ \mathtt{else}\ \sigma_1)$ [BR (c), (j)]

**Figure 2: Proof with program-capturing answer literals.**

using the $\mathtt{if-then-else}$ constructor. Here we assume that $A$ represents a computable predicate. We have a different rule for the case when $A$ is not computable but it is not included here for the lack of space. Note that this rule (and all other rules of this section) are sound, that is, the conclusion is a logical consequence of the premises.

We also modify the three superposition inference rules of Figure 1 analogously:

$$\frac{l = r \vee C \vee \mathsf{ans}(\overline{r_1}) \quad L[l'] \vee D \vee \mathsf{ans}(\overline{r_2})}{(L[r] \vee C \vee D \vee \mathsf{ans}(\mathtt{if}\ l = r\ \mathtt{then}\ \overline{r_2}\ \mathtt{else}\ \overline{r_1}))\theta}$$

$$\frac{l = r \vee C \vee \mathsf{ans}(\overline{r_1}) \quad s[l'] \neq t \vee D \vee \mathsf{ans}(\overline{r_2})}{(s[r] \neq t \vee C \vee D \vee \mathsf{ans}(\mathtt{if}\ l = r\ \mathtt{then}\ \overline{r_2}\ \mathtt{else}\ \overline{r_1}))\theta}$$

$$\frac{l = r \vee C \vee \mathsf{ans}(\overline{r_1}) \quad s[l'] = t \vee D \vee \mathsf{ans}(\overline{r_2})}{(s[r] = t \vee C \vee D \vee \mathsf{ans}(\mathtt{if}\ l = r\ \mathtt{then}\ \overline{r_2}\ \mathtt{else}\ \overline{r_1}))\theta}$$

With these modifications, all clauses generated in saturation contain at most one answer literal. The proof search successfully finishes if it obtains a clause only containing an answer literal $\mathsf{ans}(\overline{r})$. In that case we postprocess $\overline{r}$ by replacing all free variables $z$ by new constants $c_z$ of suitable types (representing an arbitrary value) and the skolem constants $\overline{\sigma}$ by the corresponding variables $\overline{x}$ from the input formula, and thereby obtain the final synthesized program.

Note that in order to produce a computable program, all terms substituted for the variables $\overline{y}$ need to be computable.

*Example 4.1.* Consider the problem of synthesizing the maximum function for two integer arguments, specified by the following formula:

$$\forall x_1, x_2 \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\big(y \geq x_1 \wedge y \geq x_2 \wedge (y = x_1 \vee y = x_2)\big) \quad (2)$$

The corresponding derivation using theory axioms of $\mathbb{Z}$ is displayed in Figure 2. The clauses (a) and (b) are the clausified skolemized negation of (2) with added answer literal, with skolem constants $\sigma_1, \sigma_2$ corresponding to $x_1, x_2$, respectively. Clauses (c) and (d) are the irreflexivity and transitivity axioms for $<$, respectively. We obtain (e) and (f) by equality resolution of (a) and (b) with substitutions $y \mapsto \sigma_1$ and $y \mapsto \sigma_2$, respectively. These two inferences instantiate the $y$ in the answer literal. Clauses (g) and (h) result from binary resolution of (c) with (e) and (c) with (f), respectively. We then obtain clause (i) by binary resolution of (d) and (h). Then we apply the modified binary resolution with answer literals on (g) with (i), which results into (j) containing an answer literal with $\mathtt{if-then-else}$ combining the program fragments from the answer literals of (g) and (i). In the final step we apply one more binary resolution on (c) and (j), obtaining a clause containing the answer literal $\mathsf{ans}(\mathtt{if}\ \sigma_1 < \sigma_2\ \mathtt{then}\ \sigma_2\ \mathtt{else}\ \sigma_1)$. By replacing the skolem constants $\sigma_1$ and $\sigma_2$ by $x_1$ and $x_2$, we obtain the realizer $f$ defined as

$$f(x_1, x_2) = \mathtt{if}\ x_1 < x_2\ \mathtt{then}\ x_2\ \mathtt{else}\ x_1.$$

## 4.1 Implementation Prototype

We implemented our new inferences rules, in the theorem prover Vampire. Besides adding answer literal versions of the binary resolution and superposition rules, we also modified the unit resulting resolution of Vampire. Many other rules of Vampire can be reduced to a combination of the rules from Figure 1. However, for the sake of efficiency and for making program synthesis practical, in future we should implement answer literal variants of all, or the majority of, Vampire's rules.

Our implementation prototype, consisting of approximately 530 lines of C++ code, is available at https://github.com/vprover/vampire/tree/hzzv-answer-lits-newfn. The synthesis functionality can be turned on using the option `--question_answering_synthesis`.

In addition to the maximum function, our implementation can synthesize the identity function, exclusive disjunctive answers or a solution to a simple polynomial equation, as well as some simple benchmarks from SyGus competition [1][2] (after translating them into forall-exists formulas).

# 5 CONCLUSIONS AND FUTURE WORK

We synthesise recursion-free programs using superposition theorem proving. We extend question answering to handle complex forall-exists queries over computable programs. Synthesizing computable programs when proofs can contain both computable and non-computable functions and predicates can also be done but is not included here.

The most challenging task is extending our synthesis method to recursive programs and proofs using induction. We also aim to integrate answers literal into the AVATAR framework [18] and simplify synthesized programs by post-processing.

# ACKNOWLEDGMENTS

This work was partially funded by the ERC CoG ARTIST 101002685 and the FWF grant LogiCS W1255-N23.

# REFERENCES

[1] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. In *SYNT@CAV*. 97–115.

[2] Simon Cruanes. 2017. Superposition with Structural Induction. In *FroCoS*. 172–188.

[3] Cordell Green. 1969. Theorem-Proving by Resolution as a Basis for Question-Answering Systems. *Machine Intelligence* (1969), 183–205.

[2] Namely fg_array_search_2, fg_array_sum_2_5, and fg_fivefuncs.sl of the CLIA track.

[4] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. 2020. Induction with Generalization in Superposition Reasoning. In *CICM*. 123–137.
[5] Márton Hajdu, Petra Hozzová, Laura Kovács, and Andrei Voronkov. 2021. Induction with Recursive Definitions in Superposition. In *FMCAD*. 1–10.
[6] Petra Hozzová, Laura Kovács, and Andrei Voronkov. 2021. Integer Induction in Saturation. In *CADE*. 361–377.
[7] S.C. Kleene. 1945. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic* 10 (1945), 109–124.
[8] Laura Kovács, Simon Robillard, and Andrei Voronkov. 2017. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 260–270. https://doi.org/10.1145/3009837.3009887
[9] Laura Kovács and Andrei Voronkov. 2009. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5503)*, Marsha Chechik and Martin Wirsing (Eds.). Springer, 470–485. https://doi.org/10.1007/978-3-642-00593-0_33
[10] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *CAV*. 1–35.
[11] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121.
[12] R. Nieuwenhuis and A. Rubio. 2001. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasonings*. Vol. I. 371–443.
[13] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. 2022. Satisfiability and Synthesis Modulo Oracles. In *VMCAI*. 263–284.
[14] Giles Reger. 2018. Revisiting Question Answering in Vampire. In *Vampire Workshop*. 64–74.
[15] Giles Reger and Andrei Voronkov. 2019. Induction in Saturation-Based Proof Search. In *CADE*. 477–494.
[16] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. (2006), 404–415.
[17] Tanel Tammet. 1994. Completeness of Resolution for Definite Answers with Case Analysis. In *CSL*. 309–323.
[18] Andrei Voronkov. 2014. AVATAR: The Architecture for First-Order Theorem Provers. In *CAV*. 696–710.