Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

A comparative study of methods for recursion-free program synthesis

Bachelor Thesis

2024
Richard Steven Žilinčík

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

A comparative study of methods for recursion-free program synthesis

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	Mgr. Petra Hozzová

Bratislava, 2024
Richard Steven Žilinčík

73376257

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Richard Steven Žilinčík
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: A comparative study of methods for recursion-free program synthesis
Komparatívna štúdia metód na syntézu programov bez rekurzie

Anotácia: Syntéza programov je oblasť formálnych metód zaoberajúca sa automatickou konštrukciou programov spĺňajúcich danú špecifikáciu. Tieto špecifikácie môžu mať rôzne podoby: napríklad funkčná špecifikácia vyjadrená logickou formulou, alebo množina párov vstupov a výstupov, alebo program, ktorý má byť zoptimalizovaný, atď. Navyše, rôzne metódy syntetizujú programy z rôznych tried - niektoré slúžia na vypĺňanie medzier v šablónach, niektoré syntetizujú programy bez rekurzie a cyklov, iné programy s rekurziou alebo cyklami.
Táto práca by mala preskúmať metódy syntetizujúce programy bez rekurzie a cyklov na základe funkčnej špecifikácie zadanej logickou formulou - najmä framework od Mannu a Waldingera pre deduktívnu syntézu [1], metódu založenú na vyvrátení negácie špecifikácie používajúcu SMT-solvery od Reynoldsa et al. [2], a metódu založenú na saturačnom algoritme od Hozzovej et al. [3].

Cieľ: Cieľom práce je porovnať metódy, ktoré syntetizujú programy na základe logickej špecifikácie vstupno-výstupnej relácie. Tieto metódy majú spravidla špecifikáciu zadanú formulou v predikátovej logike prvého rádu, prípadne rozšírenú o syntaktické obmedzenia hľadaného programu. Metódy by boli porovnávané z hľadiska (1) expresivity špecifikácie v kontexte sily logiky a syntaktických obmedzení, (2) konkrétnych príkladov, ktoré tieto metódy (ne)vedia skonštruovať (teoreticky, a pre metódy, ktoré majú aj spustiteľnú implementáciu, aj prakticky).

Literatúra: [1] Manna, Z., and Waldinger, R. "A deductive approach to program synthesis." ACM Transactions on Programming Languages and Systems (TOPLAS) 2.1 (1980): 90-121.
[2] Reynolds, A., et al. "Refutation-based synthesis in SMT." Formal Methods in System Design 55 (2019): 73-102.
[3] Hozzová, P., et al. "Program synthesis in saturation." Automated Deduction (CADE) 29 (2023): 307-324.

Vedúci:	Mgr. Petra Hozzová
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.

73376257

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátum zadania: 06.10.2023

Dátum schválenia: 06.10.2023

doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

....
študent

....
vedúci práce

73376257

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Richard Steven Žilinčík
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: A comparative study of methods for recursion-free program synthesis

Annotation: Program synthesis is an area of formal methods that focuses on automatically constructing a program satisfying the given specification. The specifications come in many shapes and flavors: e.g., a functional specification expressed as a logical formula, a set of input-output examples, a program to be optimized, etc. Further, different methods synthesize programs from different classes - some fill holes in given templates, some synthesize recursion- and loop-free programs, and some synthesize programs using recursion or loops.
This thesis should explore the methods that synthesize recursion-free (and loop-free) programs from functional specifications expressed as logical formulas - mainly Manna and Waldinger's framework for deductive program synthesis [1], refutation-based method utilizing SMT-solving from Reynolds et al. [2], and a saturation-based method from Hozzová et al. [3].

Aim: The goal of the thesis is to compare methods for recursion-free program synthesis with specification given as a logical formula capturing the input-output relation. These methods usually take a specification given as a formula in first-order logic, possibly extended by syntactic restrictions on the target program. The thesis should compare the following aspects of the methods: (1) expressivity of the specification in the context of the power of the logical language and the syntactic restrictions, (2) specific examples that can(not) be constructed using these methods (in theory, and also in practice for those methods, that have a maintained implementation).

Literature: [1] Manna, Z., and Waldinger, R. "A deductive approach to program synthesis." ACM Transactions on Programming Languages and Systems (TOPLAS) 2.1 (1980): 90-121.
[2] Reynolds, A., et al. "Refutation-based synthesis in SMT." Formal Methods in System Design 55 (2019): 73-102.
[3] Hozzová, P., et al. "Program synthesis in saturation." Automated Deduction (CADE) 29 (2023): 307-324.

Supervisor:	Mgr. Petra Hozzová
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.

73376257

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

Assigned:	06.10.2023	
Approved:	06.10.2023	doc. RNDr. Dana Pardubská, CSc. Guarantor of Study Programme

....	....
Student	Supervisor

Abstrakt

Syntéza programov je proces konštruovania korektného programu podľa vstupnej špecifikácie. Porovnávame tri metódy syntézy programov bez rekurzie zo špecifikáií v rozšíreniach prvorádovej logiky. Porovnávané metódy sú najprv stručne predstavené. Jazyky vstupnej špecifikácie sú porovnané podľa expresivity, postupy syntézy sú vyhodnotené podľa použiteľnosti na triedach problémov, a na koniec sú implementácie metód otestované na súbore referenčných problémov.

Kľúčové slová: syntéza programov, automatizované dokazovanie

Abstract

Program synthesis is the process of constructing a correct program according to an input specification. We compare three methods of synthetizing recursion-free programs from specifications given in extensions of first-order logic. The compared methods are first each briefly introduced. The input specification languages are compared by expressivity, the synthesis procedures are evaluated by their usability and aplicability to classes of problems, and finally, implementations of the methods are tested against a set of benchmarks.

Keywords: program synthesis, automated reasoning

Contents

List of Figures

Introduction

Program synthesis is a sub-field of computer science that studies processes of taking a formal specification, and outputting a program that meets the specification. There are a number of common specification formats, including logical formulas, examples of input-output pairs, or an existing program for which we want to find an equivalent, better optimized program. The output of the procedures may also come in different forms or be limited in certain ways, such as filling in blanks in user supplied templates, or programs limited to recursion-free constructs. The goal of program synthesis is to reduce the potential for human error and necessary effort to produce a correct program, by partially or fully automating the task of programming. Program synthesis combines techniques from formal methods, programming language theory, and artificial intelligence to achieve its goals.

In this thesis, we focus on the sub-field of deductive synthesis. Deductive synthesis methods are closely related to automated theorem proving and belong to the oldest branches of program synthesis [8]. This approach assumes a complete formal specification of the desired behavior, in the form of a pre-condition/post-condition pair in a formal language. A proof of the post-condition is constructed from the pre-condition, and the output program is extracted from this proof. By construction, this program is guaranteed to satisfy the original specification. This is in contrast to other branches of program synthesis, where the specification may be imprecise or incomplete, and a separate verification step may be required to ensure that the program really complies with the specification.

The main goal of our work is to compare the strengths and limitations of the methods by Manna and Waldinger [19], Reynolds et al. [24], and Hozzová et al. [16]. Previous works have conducted general surveys of program synthesis [12, 9, 10]. Among work that focuses on comparison and analysis, reports from Syntax-Guided Synthesis Competition (SyGuS-Comp) [2, 3] analyze from a high level view results of many implementations supporting SyGuS specifications. To the best of our knowledge, there is no work attempting specifically to study the set of methods we have chosen and their relative weaknesses and strengths in depth.

The first chapter briefly defines concepts used throughout the rest of the work. Chapter 2 provides a self-contained introduction to all of the compared methods. Chapter 3 compares the methods by their input specification languages, synthesis of an example program, and the strengths and weaknesses of each method. In Chapter 4, implementations of the methods are tested against a set of benchmarks and the results are evaluated.

Chapter 1

Preliminaries

Throughout the thesis, we work with standard first-order logic, using the standard notions of terms, literals, interpretations, formulas, and related concepts. We further explain selected terminology used in the remainder of the work and for details we refer to /citekovacs13.

An *expression* is either a formula or a term, and $E[t]$ denotes a expression E containing the subexpression t. Subsequent uses of $E[s]$ then denote the expression E with all instances of t replaced with the subexpression s. Further, to emphasize the case when the term t is a variable x, we sometimes write $E[x \leftarrow s]$ for $E[s]$.

Bold variable and constant names denote a vector of variables or constants respectively. Eg. $\boldsymbol{x}$ stands for a vecotor $x_1, x_2, \ldots, x_k$

Single-invocation property Any quantifier-free first-order formula in the form of $Q[\boldsymbol{x}, y]$ obtained from a formula $Q[\boldsymbol{x}, f(\boldsymbol{x})]$ by replacing $f(\boldsymbol{x})$ with y.

Clausal normal form (CNF) A formula is in CNF if it is a conjunction of clauses, where clause means a disjunction of literals.

Skolemization A formula is in Skolem normal form if all of its quantifiers are universal, and appear before the rest of the formula. Any first-order formula can be Skolemized without affecting satisfiability.

In many sorted logics, a term is said to be of type `bool` if it will evaluate to either `true` or `false`.

Completeness A formal system is called complete if every true formula can be derived through the system.

A *Unifier* θ of terms A and B is a substitution such that $A\theta$ and $B\theta$ are syntactically equal.

The *Most general unifier* (MGU) θ is a unifier of A and B, such that for each unifier of A and B σ there exists a unifier γ, such that $\theta = \sigma\gamma$.

Superposition calculus $\mathbb{S}$up A logical inference system. It is parametrized by a *simplification ordering* $\succ$ on terms, and a *selection function*, which selects a subset of literals in each clause (we shall denote selected literals of a clause by underlining them). As long as the selection function conforms to certain properties, $\mathbb{S}$up is both *sound* and *refutationally complete* [18]. We display $\mathbb{S}$up in Figure 1.1.

Superposition:

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

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

Binary resolution:

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

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

Factoring:

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

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

Equality resolution:

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

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

Equality factoring:

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

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

Figure 1.1: Superposition calculus $\mathbb{S}$up. The underlined literals are selected.

Chapter 2

Introduction of the Compared Methods

In this chapter, we present a short introduction to each of the compared methods as they have been described in previous publications.

2.1 Overview

Across all of the compared methods, the input is given in some extension of first-order logic, and the synthesized programs are expressed in first-order logic extended by if-then-else constructors.

The method of Manna et al. [19] is in principle capable of constructing programs involving recursion, though for the purpose of our work, we will only be interested in non-recursive programs that could be generated. It is described as combining techniques of unification, mathematical induction, and transformation rules into a single deductive system. It describes sequents (data structures holding conditions), and a set of transformation rules over these sequents. Once the sequents conform to a final form, an executable program can be extracted from them. Notably, no implementation is provided in this paper, and no algorithm is given for directing the application of transformation rules. Thus for this approach, we can not run automated tests to compare with the remaining approaches and must attempt to deduce possible results.

The work of Reynolds et al. [24] is significant for being the first program synthesis engine implemented inside an SMT solver. Previous approaches utilizing SMT and SAT solvers did not directly formulate the synthesis problem as a theorem proving task, but used logic implemented outside of the theorem prover to query the prover multiple times. Usually, the role of the theorem prover was to find counterexamples or prove the correctness of candidate program fragments. Reynolds et al. formulate the whole synthesis problem as a theorem proving problem, and solve it within the framework of an SMT solver. This is done by turning the constraints into a universally quantified formula, and extracting the program from a proof of unsatisfiability of its negation. Counterexample-guided techniques are used for instantiating quantifiers. A method of encoding syntactic restrictions is also presented. The algorithms described are implemented inside the CVC4 SMT solver [6], and later in its successor cvc5 [5], allowing us to run practical tests on them.

The method of Hozzová et al. [16] uses first-order theorem proving to extract code from correctness proofs of specifications. The input specification language is extended with the ability to mark specific symbols as uncomputable, and forbid their use in the output program. A modified version of the superposition calculus [4] is defined and proven sound. The program is constructed while deriving a proof of the specification using the modified superposition calculus. Substitutions into terms are tracked using answer literals [11], and eventually composed to form the final program. This method is implemented within the Vampire theorem prover [18].

2.2 Deductive Method of Manna and Waldinger

The work of Manna and Waldinger [19] is among the earliest focusing on the problem of program synthesis and predates the work of both Reynolds et al. [24] and Hozzová et al. [16] by more than three decades. It describes not yet a complete solution, but a framework within which different theorem proving algorithms might be used to guide the synthesis. In the remainder of this thesis we will be referring to this method as *the deductive method.*

2.2.1 Specification

The program specification names a function to synthesize and its list of arguments. Restrictions may be given in the form of pre-conditions for the function's arguments, and post-conditions on the arguments and outputs. An example specification might look as follows:

$$(1)\ f(a) = \texttt{find}\ z\ \texttt{such that}\ R(a,z)\ \texttt{where}\ P(a)$$

where f is the function to synthetize, a is the list of arguments, z is the list of outputs, P is the pre-condition, and R is the post-condition. The specification describes a functional program, with no side effects. For such a specification, the framework requires the use of a theorem proving algorithm to prove the following conjecture within the framework.

$$(2)\ \forall a. \exists z. P(a) \Rightarrow R(a,z)$$

2.2.2 Basic Structure

The datastructure operated on during the course of the proof is a *sequent*, which we will represent as a table. Each line of the sequent contains either one assertion or one goal, and optionally corresponding output expression. The pre-condition from the specification is added as an assertion, and the post-condition as a goal, with the output variable as its output expression. Formula (1) is rewritten as:

assertions	goals	outputs
$P(a)$		
	$R(a, z)$	z

If during the course of the proof, by applying transformations, we reach the assertion $false$, this is equivalent to the precondition in (2) being false, and therefore the whole synthesis conjecture being true. If the assertion line has a corresponding output expression, this, then, is a program which satisfies the given specification. Similarly, if during the course of the proof we reach the goal *true*, this corresponds to the right side of the implication in (2) being true, and therefore also the whole conjecture being true, and the corresponding output expression is the synthetized program.

The distinction between assertions and goals exists only for ease of understanding and does not add to the power of the framework. Any goal can be replaced with an assertion containing its negation and vice-versa.

At the beginning of the proof, except for the given pre- and post-condition, the sequent may also contain axioms of the theory being reasoned in in the assertions column without output expressions. This is one of two ways by which domain specific rules are injected into the framework's reasoning, the other being the transformation rules described later. The proof proceeds by applying transformations onto the sequent to produce new lines, with the aim of reaching the assertion $false$, or the goal *true*.

2.2.3 Rules

The notation we will use for demonstrating rules in this section will be a sequent divided by a double horizontal line, with the premises above the double line, and the consequents below it.

Splitting Rules

The *andsplit* rule allows the decomposition of an assertion in the form of a conjunction into separate assertions, and adds them to the end of the sequent with the same output expression.

assertions	goals	outputs
$F \wedge G$		t
F		t
G		t

The *orsplit* rule works analogically for goals in the form of disjunctions.

assertions	goals	outputs
	$F \vee G$	t
	F	t
	G	t

The *ifsplit* rule applies to goals in the form of an implication. The left side of the implication is appended as an assertion, the right side of the implication as a goal, both with an unchanged output expression.

assertions	goals	outputs
	$F \Rightarrow G$	t
F		t
	G	t

Transformation Rules

Transformation rules are the second way in which domain specific reasoning enters the framework. Transformations act on both assertions and goals, but in different ways. Let

$$(3)\ r \rightarrow s \texttt{ if } P$$

be a transformation rule and F an assertion containing a subformula r'. If there exists a unifier θ such that $r\theta$ is identical to $r'\theta$, (3) may be applied to F, producing an assertion

$$P\theta \Rightarrow F\theta[r'\theta \leftarrow s\theta]$$

with the output $t\theta$. Similarly, let (3) be a transformation rule and G a goal containing a subformula r'. If there exists a unifier θ such that $r\theta$ is identical to $r'\theta$, (3) may be applied to G, producing a goal

$$P\theta \wedge G\theta[r'\theta \leftarrow s\theta]$$

with the output $t\theta$.

To retain soundness in the proof, in all transformation rules used, $P \Rightarrow r = s$ must hold in the underlying theory.

Resolution Rules

The resolution rules are a generalization of the original resolution principle [25] which applies without needing to operate on clauses in conjunctive normal form. Let F and G be assertions within a sequent, let P_1 and P_2 be subformulas of F and G respectively. Let θ be the most general unifier of P_1 and P_2.

The AA-resolution rule states that a new assertion can be appended in the following way.

assertions	goals
F G	
$F\theta[P_1\theta \leftarrow true] \vee G\theta[P_2\theta \leftarrow false]$	

Since goals can be seen as negated assertions, we also have GG-resolution

assertions	goals
	F G
	$F\theta[P_1\theta \leftarrow true] \wedge G\theta[P_2\theta \leftarrow false]$

GA-resolution

assertions	goals
F	 G
	$F\theta[P_1\theta \leftarrow true] \wedge \neg G\theta[P_2\theta \leftarrow false]$

and AG-resolution.

assertions	goals
 G	F
	$\neg F\theta[P_1\theta \leftarrow true] \wedge G\theta[P_2\theta \leftarrow false]$

For all of the resolution rules, the output expression behaves in the same way. If only one of the used lines had an output expression t, then the resulting line has the output expression $t\theta$. If the first line had the output expression t_1 and the second line had the output expression t_2, the resulting output expression will be

$$\texttt{if } P_1\theta \texttt{ then } t_1\theta \texttt{ else } t_2\theta$$

2.2.4 Polarity

The polarity strategy attempts to restrict rule applications which would not be helpful to proving the conjecture. We assign a *polarity* to every subformula of a sequent as follows

1. each goal is positive
2. each assertion is negative
3. if a subformula S has form "$\neg\alpha$," then its component α has a polarity opposite to S
4. if a subformula S has form "$\alpha\wedge\beta$", "$\alpha\vee\beta$", "$\forall x.\alpha$", or "$\exists x.\beta$," then its components α and β have the same polarity as S
5. if a subformula C has form "$\alpha \Rightarrow \beta$," then β has the same polarity as S, but α has the opposite polarity

All rule applications which would substitute a subformula for $true$ or $false$ are subject to restrictions. A subformula can only be replaced by $true$ if it has at least one positive occurrence in the assertion or goal, by $false$ if it has at least one negative occurrence. The framework remains complete in first-order logic when restricted by the polarity strategy. In practice however, this strategy alone is insufficient for pruning the proof search space to a size which can be searched effectively.

2.3 SMT-Based Method of Reynolds et al.

Reynolds et al. [24] describe several related techniques for integrating program synthesis into a pre-existing SMT-solver. Some have proven unpractical through experimentation, and to keep the scope of this section reasonable, we will only describe two which have been implemented and proven practical. In the remainder of this thesis, we will refer to this method as *the SMT-based method.*

The first is specialized for conjectures that can be expressed as *singe-invocation properties*. The second is more generic, albeit its heuristics for finding candidate solutions are somewhat weaker.

In this section, we'll use the following specification formula as an example:

$$\exists f.\forall x_1, x_2.\ f(x_1,x_2) \geq x_1 \wedge f(x_1,x_2) \geq x_2 \wedge (f(x_1,x_2) = x_1 \vee f(x_1,x_2) = x_2) \quad (2.1)$$

2.3.1 Specification

cvc5 takes a specification in a standardized language developed for this task in coordination by multiple teams: SyGuS (Syntax Guided Synthesis) [21]. SyGuS is unique in its ability to place complex syntactic restrictions on accepted solutions in addition to the usual semantic restrictions. Given that the language described by the syntactic restrictions contains efficient solutions for the program requested to be synthetized and depending on the specific synthesis strategy selected, the restrictions can also *guide the solver towards a solution.*

The semantic restrictions take the form of a synthesis conjecture

$$\exists f_1, \ldots, f_n . \forall v_1, \ldots, v_m . \alpha \implies \varphi,$$

where $f_1, \ldots, f_n$ are the functions to be synthetized, and the assumptions α and the constraints φ are lists of terms of sort `bool`.

The syntactic restrictions are provided as a context-free formal grammar G. Each rule must have a type τ associated with it. On the left side of the i-th rule is a variable v_i of type τ_i. Symbol y_1 of type τ_1 is referred to as the *start symbol* of G. The type τ_1 must always be equal to the type of the function to synthetize. The expressions on the right-hand side of the rule may be either (`Variable` σ_v), allowing rewriting to any variable of type σ_v, (`Constant` σ_c), allowing rewriting to any constant of type σ_c, or an ordinary term of type τ_i.

2.3.2 Single-Invocation Properties

The first technique works by first finding a solution disregarding the syntactic restrictions, and then attempting to reconstruct it to fit them.

Let $Q[\boldsymbol{k}, y]$ be the skolemized single-invocation form of formula 2.1, where $\boldsymbol{x}$ has been replaced by fresh constants $\boldsymbol{k}$. The algorithm builds up a set Γ of ground instances of $\neg Q[\mathbf{k}, y]$, and then uses its contents to build the first version of the program. The set Γ begins as an empty set. Then, while Γ is satisfiable, the SMT-solver looks for a model I of Γ satisfying $Q[\mathbf{k}, \mathtt{e}]$, where $\mathtt{e}$ is a fresh uninterpreted constant. If such a model is not found, the algorithm terminates without a solution. If such a model is found, the ground instance $\neg Q[\mathbf{k}, t[\mathbf{k}]]$ for some term $t[\mathbf{x}]$ such that $t[\mathbf{k}]^{\mathcal{I}} = \mathtt{e}^{\mathcal{I}}$ is added to Γ.

If Γ becomes unsatisfiable, the loop terminates and the algorithm proceeds to the next step. Here the first version of the program is constructed out of the elements of Γ. The program always has the form of a nested `if-then-else` statement. Assuming $\{\neg Q[\mathbf{k}, t_1[\mathbf{k}]], \ldots, \neg Q[\mathbf{k}, t_p[\mathbf{k}]]\}$ is an unsatisfiable subset of Γ, the solution returned is

$$\lambda \boldsymbol{x}.\mathtt{if}\ Q[\mathbf{k}, t_p[\mathbf{k}]]\ \mathtt{then}\ t_p[\mathbf{k}]\ \mathtt{else}\ (\cdots \mathtt{if}\ Q[\mathbf{k}, t_2[\mathbf{k}]]\ \mathtt{then}\ t_2[\mathbf{k}]\ \mathtt{else}\ t_1[\mathbf{k}] \cdots).$$

This solution satisfies the semantic constraints, but not necessarily the syntactic ones. Many SMT-solvers (cvc5 included) can reason about the theory of algebraic datatypes. The syntactic restrictions can be embedded into restrictions over datatypes, thus enabling the solver to reason about them. For example the syntactic constraint

$$(\text{Int} \rightarrow x_1 \mid x_2 \mid 0 \mid 1)$$

which allows only the variables x_1, x_2 and the literals $0, 1$ to appear as integers in the program, can be encoded as a datatype

$$\mathsf{I} := \mathsf{x}_1 \mid \mathsf{x}_2 \mid \mathsf{zero} \mid \mathsf{one}.$$

The conversion from the grammar to the datatypes can be automated.

To tie the intended semantics of the symbols to the datatypes, *evaluation operators* are defined. Evaluation operators take as arguments a datatype instance encoding a term, and the original input variables. They evaluate the datatype to its counterpart in the regular logic language of the prover. For example,

$$\mathsf{ev}(\mathsf{plus}(\mathsf{one}, \mathsf{one}), \boldsymbol{x}) = 2.$$

After finding a candidate solution regardless of the syntactic restrictions, an attempt to reconstruct the candidate solution to fit the syntactic restrictions starts. The algorithm maintains a set A of triplets $(u\downarrow, u, \delta)$, where δ is a datatype, u is a term of type δ constrained by the corresponding restrictions for that type, and $u\downarrow$ is a normalized term equivalent to u. The algorithm begins by finding all subterms of the candidate solution which do not satisfy their respective restrictions. For each of these subterms t, it calls a function rcon, which checks the set A for a triplet containing a normalized term $u\downarrow$ equal to t normalized $t\downarrow$. If such a triplet is found, $t\downarrow$ is replaced by u. If no such triplet is found, rcon attempts to substitute the term for a function call, if the restrictions allow it, and solve the problem recursively over the terms given as arguments of the function call. If this is not possible, rcon returns without making progress, and the main loop of the algorithm adds new triplets to A and tries again.

2.3.3 Other Properties

If the property can not be rewritten as single-invocation, an alternative technique is used. The synthesis conjecture is rewritten, so that each invocation of the function is replaced with the evaluation of a term constructed from the custom datatypes. E.g. the formula 2.1 can be rewritten as

$$P_{\mathsf{ev}}[g, \boldsymbol{x}] := \mathsf{ev}(g, \boldsymbol{x}) \geq x_1 \wedge \mathsf{ev}(g, \boldsymbol{x}) \geq x_2 \wedge (\mathsf{ev}(g, \boldsymbol{x}) = x_1 \vee \mathsf{ev}(g, \boldsymbol{x}) = x_2).$$

The solver then attempts a refutation of $\forall g \exists \boldsymbol{x} \neg P_{\mathsf{ev}}[g, \boldsymbol{x}]$. The algorithm maintains a set Γ of counterexample inputs, and a limit on the size of candidate solutions n, which starts as 1. It proceeds by generating solutions within the size limit which are not invalidated by any of the collected counterexamples. If such a candidate is found, the solver tries to find counterexample inputs to invalidate the new solution. If no counterexample is found, the new solution is returned. Otherwise, the new counterexample is added to Γ, and the algorithm loops back to looking for candidate solutions. If none can be found within the size limit n, n is increased.

In this case, a potential solution is instantiating g with $\mathsf{if}(\mathsf{le}(\mathsf{x}_1, \mathsf{x}_2), \mathsf{x}_2, \mathsf{x}_1)$, which can be rewritten back into the solver's language as `if` $x_1 \leq x_2$ `then` x_2 `else` x_1.

2.4 Saturation-Based Method of Hozzová et al.

The work of Hozzová et al. is the most recent among the methods described here. It too makes use of automated proving to facilitate synthesis, and does so by integrating the process into a first-order logic theorem prover, Vampire [18]. In the remainder of this thesis, we will refer to this method as *the saturation-based method.*

2.4.1 Specification

The specification is given in a superset of the SMT-LIB language [TODO: citation]. It is extended to allow declaring a function to synthetize with its argument list, and optionally a list of so-called *uncomputable* symbols. These are symbols which are not allowed to appear in the output program. If this list is not given, all symbols are assumed to be computable. Conditions and assertions from the specification are then collected and processed into a conjecture of the form

$$A_1 \wedge \dots A_n \Rightarrow \forall \boldsymbol{x}. \exists y. F[\boldsymbol{x}, y]$$

where $A_1 \wedge \dots A_n$ are the assumptions, and F contains all of the conditions.

2.4.2 Saturation and Superposition

Saturation-based proving attempts a proof by refutation. It attempts to refute the negation of the conjecture to-be-proven based on the assertions. The proof proceeds by negating the conjecture and converting all the formulas into CNF, and iteratively computing *consequences* of these clauses. Consequences are derived by applying rules from an inference system onto already derived clauses.

A commonly chosen inference system is the *superposition calculus* $\mathbb{S}$up, described in Chapter 1, which is also used by Vampire.

2.4.3 Superposition with Answer Literals

Superposition (Sup):

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

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

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

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

Binary resolution (BR):

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

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

Factoring (F):

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

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

Equality resolution (ER):

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

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

Equality factoring (EF):

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

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

Figure 2.1: Selected rules of the extended superposition calculus $\mathbb{S}$up for reasoning with answer literals, with underlined literals being selected.

Similarly to output expressions in Subsection 2.2, the system uses *answer literals* [11] to track partial progress during the synthesis of the program. They capture candidate program fragments associated with each clause. One answer literal is added to each clause originating from the specification conjecture. For practical reasons, these are maintained as part of the clause, as opposed to extra metadata. Whenever a substitution is performed on a clause, the change is reflected inside the answer literal. $\mathbb{S}$up is modified to take answer literals into account, but treats them differently from other literals. They are never constrained in the premises of an inference rule, which means they do not influence the process of the saturation. We display selected rules of the modified version of $\mathbb{S}$up in Figure 2.1.

2.4.4 Programs from Answer Literals

The synthesis conjecture is first skolemized and converted to CNF. Subsequently, an answer literal is inserted into each clause. The proof system is guided towards producing clauses in the form of $C[\sigma] \vee \texttt{ans}(r[\sigma])$, where both $C[\sigma]$ and $\texttt{ans}(r[\sigma])$ are computable. Here $r[\sigma]$ is a witness for y in $\exists y.F[\sigma, y]$, and such a clause means that assuming $\neg C[x]$, $r[x]$ satisfies the specification. This fact is recorded on the side, the clause is replaced by $C[\sigma]$, and the proof continues. The proof stops when the disjunction of negations of the conditions recorded becomes unsatisfiable. At that point saturation stops, and the final program is composed from segments r_i conditioned on $\neg C_i$. The structure of the program is a sequence of tests for $\neg C_i$, executing r_i if evaluated to true, and falling through to a test for C_{i+1} if false. For example, if we collected the conditions $\neg C_1$, $\neg C_2$, $\neg C_3$ with program segments r_1, r_2, r_3, respectively, and $C_1 \wedge C_2 \wedge C_3$ together with the input assertions is unsatisfiable, then the final program is constructed as

$$\texttt{if } \neg C_1 \texttt{ then } r_1 \texttt{ else if } \neg C_2 \texttt{ then } r_2 \texttt{ else } r_3.$$

2.4.5 Computable Unification

To make sure the final program only makes use of computable symbols, *computable unifiers* are used in place of MGUs. Computable unifiers can be found with a modified version of the standard unification algorithm [25]. They ensure that uncomputable symbols are never substituted into answer literals.

Chapter 3

Comparison

In this chapter we compare the previously introduced methods based on their specification language and synthesis procedure.

3.1 Specification Languages

One important differentiating factor between the methods is the strength and expressivity of the specification language. That is, what classes of problems can we specify, and how efficient and clear the encoding of the problem is. We also consider the ability to algorithmically convert from one specification system to another, which may also be useful in cases where we want to try running a problem through multiple systems to select the result that best suits us.

3.1.1 Syntactic Constraints

The deductive method does not include any dedicated mechanism for syntactic restrictions, they are altogether inexpressible in this framework.

The saturation-based method includes a weak mechanism for syntactic restrictions through the enumeration of incomputable symbols. This allows us to use symbols which are convenient for writing concise specifications, but for which we have no algorithmic implementation. Higher level restrictions on the structure of the program however are not possible.

The SMT-based method has the strongest mechanism for syntactic restrictions, based on context-free formal grammars. The set of restrictions expressable by this mechanism is clearly a superset of what can be done only by blacklisting uncomputable symbols. The grammar can also be used to express structural constraints on the program useful for ensuring properties relating to human readability, or time or memory complexity.

From this, it follows that if any syntactic restrictions are used for a specification in the saturation-based or SMT-based system, there can not exist an encoding of it in the deductive approach. Similarly, if the grammar used to specify a problem in the SMT-based method imposes non-trivial restrictions (i.e. those that do not amount to excluding a list of symbols), the specification will also be unencodable in the saturation-based approach. Conversely, since the SMT-based method's syntactic restrictions are strictly stronger than the saturation-based and deductive method's, the encodability of specifications from those systems into the SMT-based method will only be subject to the ability to convert semantic restrictions.

3.1.2 Semantic Constraints

In the saturation-based approach and the deductive approach, the constraints are given in subsets of first-order logic, possibly extended with theories. SyGuS syntax supports higher-order constructs as well, although currently the synthesis process itself will fail to produce results for them in almost all cases. While the SMT and saturation-based approaches include built-in support for multiple common theories, the deductive approach requires that they be encoded by the user in transformation rules.

Another relevant difference here between SyGuS and the input formats of the other two systems is, that as of revision 2.1, SyGuS does not have support for adding uninterpreted functions directly to the logic being used. This is possible in the superset of the SMT-LIB language that the saturation-based method uses and the deductive method.

An encoding for uninterpreted functions is possible, at the cost of higher complexity of code and scalability issues. The idea is, that since cvc5 has support for higher-order logic, it is possible to declare new universally quantified variables instead of making the function symbols a part of the logic vocabulary, and require these variables as arguments in every function that uses them in its body. This can be done in an algorithmic way.

For example, [16] used the group theory axioms:

(i) the operation $*$ is associative

(ii) the operation $*$ has a neutral element

(iii) there exists an inverse element for each element with respect to $*$.

We can define the group axioms as follows in SMT-LIB with `op` denoting $*$:

```
(declare-fun inv (s) s)
(declare-fun op (s s) s)
```

```
(declare-const e s)

; Left inverse
(assert (forall ((x s)) (= (op (inv x) x) e)))
; Left identity
(assert (forall ((x s)) (= (op e x) x)))
; Associativity
(assert (forall ((x s) (y s) (z s)) (= (op x (op y z)) (op (op x y) z))))
```

and as follows in SyGuS

```
(synth-fun f ((x s) (e s) (op (-> s s s)) (i (-> s s))) s)
(declare-var e s)
(declare-var op (-> s s s))
(declare-var i (-> s s))

(assume (forall ((x s)) (= (op (i x) x) e)))
(assume (forall ((x s)) (= (op e x) x)))
(assume (forall ((x s) (y s) (z s)) (= (op x (op y z)) (op (op x y) z))))
```

where f is the function to synthetize, and x is the original input. In this way, we allow cvc5 to use the neutral element, $*$, and the inversion operation in the construction of f even though they are not a part of the underlying theory.

The saturation-based and deductive approaches only support specifying single-invocation properties. This covers most useful properties, but does not cover some properties, like monotonicity (as it involves a relation between multiple invocations of the function).

3.2 Theoretical Demonstration

This section will demonstrate what the process of synthesis of a simple program looks like in the theoretical synthesis frameworks of each method. The running example used will be the synthesis of a program that produces the maximum of two numbers. The corresponding specification is:

$$\exists f.\forall x_1, x_2.\ f(x_1,x_2) \geq x_1 \wedge f(x_1,x_2) \geq x_2 \wedge (f(x_1,x_2) = x_1 \vee f(x_1,x_2) = x_2)$$

Since this is a single-invocation property, it can also be encoded by substituting y for each function call site.

$$\forall x_1, x_2 . \exists y.\ y \geq x_1 \wedge y \geq x_2 \wedge (y = x_1 \vee y = x_2)$$

3.2.1 Deductive Method

	assertions	goals	outputs	
1		$y \geq \sigma_1 \wedge y \geq \sigma_2 \wedge (y = \sigma_1 \vee y = \sigma_2)$	y	
2		$y \geq \sigma_1 \wedge y \geq \sigma_2 \wedge y = \sigma_1$	y	
3		$y \geq \sigma_1 \wedge y \geq \sigma_2 \wedge y = \sigma_2$	y	
4	$x \geq x$			
5		$\neg false \wedge true \wedge \sigma_1 \geq \sigma_2 \wedge \sigma_1 = \sigma_1$	σ_1	GA-resolution 4, 2
6		$\neg false \wedge true \wedge \sigma_2 \geq \sigma_1 \wedge \sigma_2 = \sigma_2$	σ_2	GA-resolution 4, 3
7		$\sigma_1 \geq \sigma_2$	σ_1	logic rules 5
8		$\sigma_2 \geq \sigma_1$	σ_2	logic rules 6
9		$\sigma_2 = \sigma_1 \vee \neg\sigma_1 \geq \sigma_2$	σ_2	logic rules 8
10		$\neg\sigma_1 \geq \sigma_2$	σ_2	orsplit 9
11		$true$	`if` $\sigma_1 \geq \sigma_2$ `then` σ_1 `else` σ_2	GG-resolution 7, 10

The above is a sequent built during the synthesis of the program. In some lines, multiple simple steps have been folded into one for the sake of brevity. The first line (1) is a skolemized version of the synthesis conjecture. Lines (2) and (3) are derived from line (1) by applying distributivity, followed by the application of the andsplit rule. Line (4) is an axiom of our logic system. Through the application of GA-resolution on lines (4) and (2), we derive line (5), using $P_1 = x \geq x, P_2 = y \geq \sigma_1, \theta = [x \leftarrow \sigma_1, y \leftarrow \sigma_1]$. Line (6) is analogical, substituting for σ_2 instead of σ_1. By applying several simple logic rules, such as simplification of $true \wedge P$ to P, and the symmetry of equality, we derive line (7) from line (5) and line (8) from line (6). The system maintains consistency if we include transformation rules from one expression to a logically equivalent expression. Line (9) is obtained by applying a transformation rule $x \geq y \Rightarrow x = y \vee \neg y \geq x$ to line (8). We obtain line (10) by applying the orsplit rule to line (9). Finally, line (11) is the result of applying GG-resolution to lines (7) and (10), with $P_1 = \sigma_1 \geq \sigma_2, P_2 = \sigma_1 \geq \sigma_2$, and $\theta = []$. This simplifies to $true \wedge \neg false$, which simplifies through a transformation rule to $true$. The final program is

$$\texttt{if } x_1 \geq x_2 \texttt{ then } x_1 \texttt{ else } x_2$$

3.2.2 SMT-Based Method

The synthesis will follow the approach outlined in section 2.3.2. First we introduce fresh constants by skolemiztion, a_1 and a_2 in place of x_1 and x_2 respectively, and e in place of y. The set Γ begins as the empty set. The first iteration of the loop finds a model $\mathcal{I}$ satisfying the conjecture. Since the conjecture is satisfied, either $\mathsf{e}^{\mathcal{I}} = {\mathsf{a}_1}^{\mathcal{I}}$ or $\mathsf{e}^{\mathcal{I}} = {\mathsf{a}_2}^{\mathcal{I}}$ must hold. Assume that in this case $\mathcal{I}$ interprets e with the same value as a_1. Then we add the formula $\neg Q[\mathbf{a}, \mathsf{a}_1]$ to Γ. In the second iteration of the loop, $\Gamma \cup Q[\mathbf{a}, \mathsf{e}]$ is determined to still be satisfiable. The next model must satisfy $\neg Q[\mathbf{a}, \mathsf{a}_1]$, which simplifies down to $\neg \mathsf{a}_1 \geq \mathsf{a}_2$. The solver's only possible choice now is to select a model where $\mathsf{e}^{\mathcal{I}} = {\mathsf{a}_2}^{\mathcal{I}}$. We add the formula $\neg Q[\mathbf{a}, \mathsf{a}_2]$, which simplifies to $\neg \mathsf{a}_2 \geq \mathsf{a}_1$, to Γ, which makes it no longer satisfiable. The procedure then terminates and builds the final solution from the elements of Γ:

$$\texttt{if } x_1 < x_2 \texttt{ then } x_2 \texttt{ else } x_1$$

3.2.3 Saturation-Based Method

As part of pre-processing, the system rewrites predicates to their canonical forms, namely in this example $x \geq y$ is rewritten to $\neg x < y$. Further, the system automatically adds relevant axioms to the search space, and produces the following derivation:

(a)	$y < \sigma_1 \vee y < \sigma_2 \vee y \neq \sigma_1 \vee \texttt{ans}(y)$	[input]
(b)	$y < \sigma_1 \vee y < \sigma_2 \vee y \neq \sigma_2 \vee \texttt{ans}(y)$	[input]
(c)	$\neg x < x$	[$<$ axiom]
(d)	$\neg x_1 < x_2 \vee \neg x_2 < x_1$	[$<$ axiom]
(e)	$\sigma_1 < \sigma_1 \vee \sigma_1 < \sigma_2 \vee \texttt{ans}(\sigma_1)$	[ER (a)]
(f)	$\sigma_2 < \sigma_1 \vee \sigma_2 < \sigma_2 \vee \texttt{ans}(\sigma_2)$	[ER (b)]
(g)	$\sigma_1 < \sigma_2 \vee \texttt{ans}(\sigma_2)$	[BR (c), (e)]
(h)	$\sigma_2 < \sigma_1 \vee \texttt{ans}(\sigma_1)$	[BR (c), (f)]
(i)	$\sigma_1 < \sigma_2$	[answer literal removal (g)]
(j)	$\sigma_2 < \sigma_1$	[answer literal removal (h)]
(k)	$\neg \sigma_2 < \sigma_1$	[BR (d), (i)]
(l)	$\square$	[BR (j), (k)]

The clauses (a) and (b) are obtained by converting the synthesis conjecture into CNF and applying skolemization to the results. Clause (c) is the axiom of irreflexivity for $<$, clause (d) is the asymmetry axiom. Clauses (e) and (f) are obtained through the application of the equality resolution rule to clauses (a) and (b), using the substitutions $[y \leftarrow \sigma_1]$ and $[y \leftarrow \sigma_2]$. Clauses (g) and (h) result from the binary resolution of clause

(c) with clause (e) and clause (c) with clause (f), respectively. Clause (g) is of the form $C[\sigma] \vee \texttt{ans}(r[\sigma])$, which means we store aside the condition $\neg\sigma_1 < \sigma_2$ and note down σ_2 as the corresponding answer program fragment. The same is done for clause (h). We remove answer literals from clauses (g) and (h), deriving (i) and (j). Through binary resolution of clauses (d) and (i), substituting $[x_1 \leftarrow \sigma_1, x_2 \leftarrow \sigma_2]$ we derive clause (k). From clauses (j) and (k) we derive the empty clause $\square$, which tells us that $\neg(\neg\sigma_1 < \sigma_2 \vee \neg\sigma_2 < \sigma_1)$ is unsatisfiable. This means we are ready to construct the final program. Either the first condition is fulfilled and we can use the corresponding program fragment, or (since we only have 2 disjuncts) we use the remaining program fragment, obtaining:

$$\texttt{if } x_1 < x_2 \texttt{ then } x_2 \texttt{ else } x_1.$$

3.3 Strengths and Weaknesses

3.3.1 Deductive Method

The deductive approach, being the oldest of the three, was developed at a time when automated theorem proving was in a much less mature state. The techniques needed for a full practical implementation were still missing in some places. As a result, the authors left significant gaps in the full algorithm to be filled in by later works. On its own, the framework described provides little guidance on how the proof search should be conducted. The only part of the system pertaining to this is the polarity strategy. It reduces the search space somewhat, but does not single out the next step to be taken, or even reduce the search space to a size that would make it practical to search by simple enumeration. In practice, this effectively makes it more akin to a framework for verification of programs. We either need to already know the proof to construct the program, or know the final program and recreate the proof working backwards.

It also contains multiple mutually redundant mechanisms. The paper introducing it itself remarks that only the resolution rules and some logical transformation rules are strictly necessary. This further enlarges the search space for possible steps without increasing the system's expressivity.

Compared to the other two methods, this framework in theory also supports the synthesis of recursive programs. The specific techniques needed for automated proof search with induction are however still an open research area, though the last decade has seen progress in integrating induction into SMT- and saturation-based reasoning [23, 22, 14], also leading to automated recursive synthesis [15].

3.3.2 SMT-Based Method

Compared to first-order theorem provers, SMT solvers excel at theory reasoning. The solvers often contain numerous heuristics for each specific implemented theory, which would be very difficult to emulate with a more generic approach. Many mature SMT-solvers have accumulated a large number of non-generalizable optimizations that provide speed-ups for specific operations in some theories, which can in practice have a significant impact on the speed of theory heavy reasoning. This is helpful for example for specifications involving non-trivial real arithmetic or reasoning about bitvectors.

3.3.3 Saturation-based Method

Even though many SMT solvers have some support for quantified formulas, in practice they often struggle as quantifiers proliferate. This is also visible in the synthesis techniques, where quantifier instantiation is often the main bottleneck. In comparison, first-order theorem provers are usually much more capable in reasoning with formulas involving multiple or alternating quantifiers, such as the previously mentioned group theory axioms.

Vampire also supplements its own strengths as a first-order theorem prover with the use of the AVATAR framework [26, 7], which queries SMT-solvers internally to aid with proof search.

Chapter 4

Experimentation

In this chapter we test the implementations of the SMT and saturation-based methods on several example specifications.

The input files used for testing are included in the electronic attachment. All tests were performed on an AMD Ryzen 5 5500U with access to 16GB of RAM, and with a 5 minute timeout. The versions used are version 1.1.0 for cvc5, and version 4.8 (commit f9cebc54f) for Vampire. cvc5 was run with default settings. Vampire was run with the following option settings:

- `--question_answering synthesis -t 300`
- `--question_answering synthesis -mode portfolio -t 300`
- `--decode lrs-11_2:1_av=off:inw=on:ile=on:irw=on:lcm=reverse:lma=on:nm=64:nwc=1.5:sp=reverse_arity:urr=on:qa=synthesis_300`

There is no maintained implementation of the deductive method, so we were unable to perform practical tests of it. The following table summarizes the evaluation results. In the rest of this chapter we describe each example in detail.

Name	Vampire	cvc5
Square of Sum	yes	no (yes with restrictions)
Absolute Value	yes	yes
Same Quotient, Different Remainder	no	yes
Invert Bitvector Addition	no	yes
Field Theory	yes	no
Quotient 1	no	no

4.1 Square of Sum

$$\exists f.\forall x_1, x_2.\ f(x_1, x_2)^2 = x_1^2 + 2x_1x_2 + x_2^2$$

This example tests whether the solvers are able to derive the identity of squaring a sum. This example is taken from [16]. It was devised by looking for a simple identity which uses an operation that can not be directly reversed on the integers.

The example was solved by Vampire and not solved by cvc5, replicating the results from [16]. In addition, we also tested the effect of adding syntactic restrictions for cvc5 to guide the synthesis process. We found that cvc5 successfully synthetized the function if it was syntactically restricted to using the input variables, 1, unary minus, addition, and multiplication. However allowing 0 or subtraction caused cvc5 to fail to solve the example within the time limit. This highlights the fragility of using syntactic restrictions in an attempt to help synthesis.

The natural solution, which was also found by vampire, is

$$f(x_1, x_2) = x_1 + x_2.$$

4.2 Absolute Value

$$\exists f. \forall x.\ f(x)^2 = x^2 \wedge f(x) \geq 0$$

The example tests whether the solvers are able to derive a function calculating the absolute value. Similarly to the previous example, the specification was written so that the function can not be expressed simply by reversing operations applied to the input.

This example was solved by both Vampire and cvc5. Perhaps an interesting observation is, that the function synthetized by cvc5 contains many redundant checks.

```
(let ((square (*x x)))
(ite (= square 0)
0
(ite (= square 4)
2
(ite (= square 1)
1
(ite (>= x 0)
x
(ite (>= x 1)
3
(* (- 1) x)))))))
```

These appear as if they could be simplified by checking for redundancy, which just have not been integrated into the synthesis process. Output minimization has not been a stated goal of any of the examined methods.

The solution to this example found by Vampire is

$$f(x) = \texttt{if } x > 0 \texttt{ then } x \texttt{ else } -x.$$

4.3 Same Quotient, Different Remainder

$$\exists f. \forall x.\ \mathsf{div}(f(x), 2) = \mathsf{div}(x, 2) \land f(x) \neq x$$

This example asks for a function that returns a number different from the input value, but with the same quotient with respect to 2. This example was solved by cvc5, and not solved by Vampire. We attribute this to the more powerful theory reasoning of the SMT solver.

The solution found by cvc5 is

$$f(x) = \texttt{if } \mathsf{div}(x, 2) = \mathsf{div}(x + 1, 2) \texttt{ then } x + 1 \texttt{ else } x - 1.$$

4.4 Invert Bitvector Addition

$$\exists f. \forall x_1, x_2 \in (\mathsf{BitVec}\ 4).\ \mathsf{bvsge}(\mathsf{bvadd}(f(x_1, x_2), x_1), x_2)$$

This example is adapted from SyGuS-Comp 2019 [1]. It tests reasoning in the theory of bitvectors [13]. It asks for a function, that given two 4-bit bitvectors, returns a 4-bit bitvector which when added to the first given bitvector, is greater than or equal to the second.

cvc5 solved this example, while Vampire does not support the theory of bitvectors, and so was unable to solve it. The purpose of this example was to show the kind of problem that SMT solvers have an inherent advantage at. Effective reasoning with the bitvector theory requires theory-specific decision procedures which first-order theorem provers generally do not implement.

The solution to this example was

$$f(x_1, x_2) = \mathsf{bvsub}(x_2, x_1).$$

4.5 Field Theory

$$\text{Assuming } (F, +, \cdot) \text{ is a field: } \exists f. \forall x_1, x_2.\ (-x_1) \cdot x_2 = -f(x_1, x_2)$$

$$\text{Assuming } (F, +, \cdot) \text{ is a field: } \exists g. \forall x_1, x_2.\ (-x_1) \cdot (-x_2) = g(x_1, x_2)$$

This example was inspired by the group theory examples from [16], but added axioms for making F a field and used different identities. The goal is to derive two basic identities, without the use of the $-$.

The encoding mentioned in section 3.1.2 was used to encode the uninterpreted functions into SyGuS for cvc5. For Vampire, the symbol for the inversion operation (unary $-$) was marked as uncomputable to make sure it is not used in the output. Like the group theory examples, this example was solved by Vampire and not by cvc5. This was expected, due to the limited ability of SMT solvers to handle quantifies, and requiring reasoning with a theory not built into the solver.

The solution to both of the above examples is

$$f(x_1, x_2) = g(x_1, x_2) = x_1 \cdot x_2$$

4.6 Quotient 1

$$\exists f. \forall x.\ x \neq 0 \Rightarrow \mathsf{div}(f(x), x) = 1$$

This example asks for an output such that its quotient with respect to the input is 1. It was selected as an example of a problem that intuitively looks like it should be trivial to solve, but at which neither solver succeeds.

It was solved by neither Vampire nor cvc5. In general, reasoning with non-linear integer arithmetic is undecidable [20], but for this specific example even the identity function fulfills the constraint. We are not entirely sure why both solvers happen to struggle with this specific case, but we speculate it may be related to general difficulties of axiomatizing integer division.

Conclusion

In this thesis, we conducted a systematic comparison of three synthesis methods for recursion-free programs. The input specification languages are compared and analyzed with respect to the limitations on the classes of programs each is capable of specifying, both in terms of syntactic and semantic restrictions on the output program. We designed an encoding for programs featuring uninterpreted functions into SyGuS through the use of higher-order variables. A direct comparison of the synthesis process of each of the three methods was carried out on a shared example. We highlighted the specific limitations of each method's synthesis process finding that the deductive method cannot be fully automated, the SMT-based method struggles with quantifiers, and the saturation-based method struggles with theory-heavy reasoning. These insights are useful for future users of the respective synthesis frameworks, as well as directing the researchers that develop them.

In the practical comparison, we designed targeted benchmarks with the intent to test hypothesized strong and weak suits of tested methods. We tested the implementations and analyzed where and why the results conformed to or diverged from our expectations.

Further work in this direction could include comparing a larger number of synthesis methods, or extending the comparison to methods for synthesis with recursion. A tool for automated bidirectional conversion between SyGuS and SMT-LIB specifications (where possible) would make it practical to conduct implementation comparisons on a larger scale. Alternative proposed specification languages, such as SemGuS [17] could be compared to the ones included here. Further comparative work could also focus on the differences in synthesized programs in terms of length or complexity.

Bibliography

[1] SyGuS-Comp 2019. `https://sygus-org.github.io/comp/2019/`. Accessed: 2024-05-23.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

[4] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[7] Nikolaj Bjøner, Giles Reger, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *2nd Global Conference on Artificial Intelligence*, pages 39–52, 2016.

[8] Rod M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

[9] Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, 2017.

[10] Pierre Flener. Achievements and prospects of program synthesis. *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part I*, pages 310–346, 2002.

[11] Cordell Green. Theorem proving by resolution as a basis for question-answering systems. *Machine intelligence*, 4:183–205, 1969.

[12] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[13] Liana Hadarean. *An efficient and trustworthy theory solver for bit-vectors in satisfiability modulo theories.* PhD thesis, New York University, 2015.

[14] Márton Hajdu, Petra Hozzová, Laura Kovács, Giles Reger, and Andrei Voronkov. Getting saturated with induction. In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, pages 306–322. Springer, 2022.

[15] Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner. Synthesis of Recursive Programs in Saturation. Technical report, EasyChair, 2024.

[16] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. Program Synthesis in Saturation. 29:307–324, 2023.

[17] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[18] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[19] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[20] Yuri V. Matiyasevich. *Hilbert's tenth problem.* MIT Press, Cambridge, MA, USA, 1993.

[21] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The SyGuS Language Standard Version 2.1. *arXiv preprint arXiv:2312.06001*, 2023.

[22] Giles Reger and Andrei Voronkov. Induction in saturation-based proof search. In *Automated Deduction–CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings 27*, pages 477–494. Springer, 2019.

[23] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 80–98. Springer, 2015.

[24] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-based synthesis in smt. *Formal Methods in System Design*, 55:73–102, 2019.

[25] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[26] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 696–710. Springer, 2014.

Appendix A: Contents of the Electronic Attachment

The electronic attachment contains the input files for the benchmarks mentioned in Chapter 4.