Theory Exploration Powered by Deductive Synthesis

Eytan Singher(✉) and Shachar Itzhaky

Technion, Haifa, Israel
{eytan.s,shachari}@cs.technion.ac.il

Abstract. This paper presents a symbolic method for automatic theorem generation based on deductive inference. Many software verification and reasoning tasks require proving complex logical properties; coping with this complexity is generally done by declaring and proving relevant sub-properties. This gives rise to the challenge of discovering useful sub-properties that can assist the automated proof process. This is known as the *theory exploration* problem, and so far, predominant solutions that emerged rely on evaluation using concrete values. This limits the applicability of these theory exploration techniques to complex programs and properties.

In this work, we introduce a new symbolic technique for theory exploration, capable of (offline) generation of a library of lemmas from a base set of inductive data types and recursive definitions. Our approach introduces a new method for using abstraction to overcome the above limitations, combining it with deductive synthesis to reason about abstract values. Our implementation has shown to find more lemmas than prior art, avoiding redundant lemmas (in terms of provability), while being faster in most cases. This new abstraction-based theory exploration method is a step toward applying theory exploration to software verification and synthesis.

Keywords: Theory exploration · Synthesis · Automatic theorem proving

1 Introduction

Most forms of software verification and synthesis rely on some form of logical reasoning to complete their task. Whether it is checking pre- and post-conditions, deriving specifications for sub-problems [1,19], or equivalence reduction [39], these methods rely on assumptions from both the input and relevant background knowledge. Domain-specific knowledge can reinforce these methods, whether via the design of a domain-specific language [29,36,45], specialized decision procedures [28], or decomposing specifications [35]. While hand-crafted techniques can treat whole classes of programs, every library or module contributes a collection

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 125–148, 2021.
https://doi.org/10.1007/978-3-030-81688-9_6

of new primitives, requiring tweaking or extending these methods. Automatic formation of background knowledge can enable effortless treatment of such libraries and programs.

In the context of verification tools, such as Dafny [27] and Leon [7], as well as interactive proof assistants, such as Coq [12] and Isabelle/HOL [33], background knowledge is typically given as a set of *lemmas*. Usually, these libraries of lemmas (*i.e.* the background knowledge) are created by human engineers and researchers who are tasked with formulating them and proving their correctness. When a proof or verification task requires auxiliary lemmas missing from the existing background knowledge, the user is required to add and prove it, sometimes repeating this process until the proof is trivial or can be found automatically. For example, both Dafny and Leon fail to prove that addition is associative and commutative from first principles—based on an algebraic construction of the natural numbers. However, when given knowledge of these properties (*i.e.* encoded as lemmas: $(x + y) + z = x + (y + z)$ and $x + y = y + x$)[1], they readily prove composite facts such as $(x + 5) + y = 5 + (x + y)$.

A possible solution is to eagerly generate valid lemmas, and to do so automatically, offline, as a precursor to any work that would be built on top of the library. This paradigm is known as *theory exploration* [8,9], and differs from the common conjecture generation approach (in theorem provers and SMT solvers [37]) that is guided by a proof goal. As opposed to using proof goal as the basis for discovering sub-goals, when eagerly generating lemmas there is a vast space of possible lemmas to consider. Currently, two main approaches exist for filtering candidate conjectures, counterexample-based and observational equivalence-based [18,22,23,43]. These filtering techniques are all based on testing and therefore require automatic creation of concrete examples.

Testing with concrete values allows for fast evaluation and filtering of terms when the data types involved are simple. However, when scaling to larger data types and function types it becomes a bottleneck of the theory exploration process. Previous research effort has revealed that testing-based discovery is sensitive to the number and size of type definitions occurring in the code base. For example, QuickSpec, which is based on QuickCheck (as are all the existing testing-based theory exploration methods), employs a heuristic to restrict the set of types allowed in terms in order to make the checker's job easier. Compound data types such as lists can be nested up to two levels (lists of lists, but not lists of lists of lists). This presents an obstacle towards scaling the approach to real software libraries, since "*QuickCheck's size control interacts badly with deeply nested types* [...] *will generate extremely large test data.*" [38]

Following are two example scenarios that attempt to represent cases from software systems where structured data types and complicated APIs exist: (i) A series of tree data-types T_i where each T_i is a tree of height i with i children of type T_{i-1}, and the base case is an empty tree. Creating concrete examples for T_i will be resource expensive, as each tree has $O(i!)$ nodes, and each node requires a

[1] In fact, these properties are hard-wired into decision procedures for linear integer arithmetic in SMT solvers.

value. (ii) An ADT (Algebraic Data Type) A with multiple fields where each can contain a large amount of text or other ADTs, and a function over A that only accesses one of the fields. Even if evaluating the function is fast, fully creating A is expensive and will impact the theory exploration run-time.

This paper presents a new symbolic theory exploration approach that takes advantage of the characteristics of induction-based proofs. To overcome the blowup in the space of possible values, we make use of *symbolic values*, which contain interpreted symbols, uninterpreted symbols, or a mixture of the two. Conceptually, each symbolic value is an abstraction representing (infinitely) many possible values. This means that preexisting knowledge on the symbolic value can be applied without fully creating interpreted values. Still, when necessary, uninterpreted values can be expanded, creating larger symbolic values, thus refining the abstraction, and facilitating the necessary computation. We focus on the formation of *equational* theories, that is, lemmas that curtail the equivalence of two terms, with universal quantification over all free variables.

We show that our symbolic method for theory exploration is more applicable and faster in many different scenarios than state-of-the-art. As an example, given standard definitions for the list functions: ++ drop take filter our method proves facts that were not found by current state-of-the-art such as:

$$(\text{take } i\ xs)\ \texttt{++}\ (\text{drop } i\ xs) = xs$$
$$\text{filter } p\ (xs\ \texttt{++}\ ys) = (\text{filter } p\ xs)\ \texttt{++}\ (\text{filter } p\ ys)$$

Main Contributions. This paper provides the following contributions:

- A system for *theory synthesis* using symbolic values to take advantage of value abstraction. Our implementation, TheSy, can discover more lemmas than were found by testing-based tools, while being faster in most cases.
- A technique to compare universally quantified terms using term rewriting techniques and a given set of lemmas, called *symbolic observational equivalence* (SOE). SOE overapproximates term equalities deducible by the given lemmas (*i.e.*, will find more equalities), thus can be used for equality reduction in context of uninterpreted values, enabling fully symbolic reasoning over a large set of terms.
- An evaluation of our theory exploration system on a set of benchmarks for induction proofs taken from CVC4 [37] and TIP 2015 [11], specifically the IsaPlanner benchmarks [21]. We compare our implementation with a current leading theory exploration system, Hipster [18], using a novel metric. This metric is insensitive to the amount of found lemmas, but rather measures their usefulness in the context of theorem proving.

2 Overview

Our theory exploration method, named TheSy (Theory Synthesizer, pronounced *Tessy*), is based on syntax-guided enumerative synthesis. Similarly to previous approaches [10,20,38], TheSy generates a comprehensive set of terms from

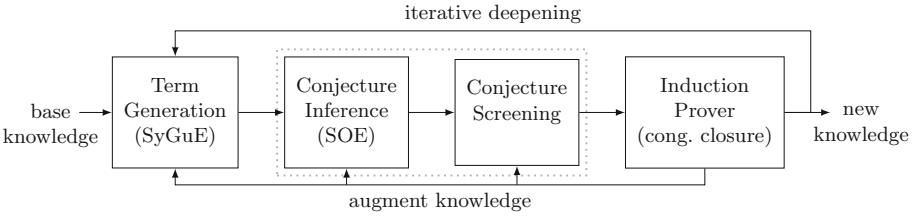

Fig. 1. TheSy system overview: breakdown into phases, with feedback loop.

the given vocabulary and looks for pairs that seem equivalent. Notably, TheSy employs deductive reasoning based on term rewriting systems to propose these pairs by extrapolating from a set of known equalities, employing a relatively lightweight (but unsound) reasoning procedure. The proposed pairs are passed as equality conjectures to a theorem prover capable of reasoning by induction.

The process (as shown in Fig. 1) is separated into four stages. These stages work in an iterative deepening fashion and are dependent on the results of each other. A short description is given to help the reader understand their context later on.

1. **Term Generation.** Build symbolic terms of increasing depth, based on the given vocabulary. Use known equalities for pruning via equivalence reduction.
2. **Conjecture Inference.** Evaluate terms on symbolic inputs, and apply deductive inference to extract new equalities, thus forming conjectures.
3. **Conjecture Screening.** Some of the conjectures, even valid ones, are special cases of known equalities or are trivially implied by them. We deem these conjectures redundant. TheSy culls such conjectures before continuing to prove the rest.
4. **Induction Prover.** The prover attempts to prove conjectures that passed screening using a normal induction scheme derived from algebraic data structure definitions in the given vocabulary. Conjectures that were successfully proven are then declared *lemmas* and added to the known equalities.

The phases are run iteratively in a loop, where each iteration deepens the generated terms and, hence, the discovered lemmas. These lemmas are fed back to earlier phases; this form of feedback contributes to discovering more lemmas thanks to several factors:

(i) Conjecture inference is dependent upon known equalities. Additional equalities enable finding new conjectures.
(ii) Accurate screening by merging equivalence classes based on known equalities.
(iii) The prover is based on known equalities with a congruence closure procedure. The more lemmas are known to the system, the more lemmas become provable by this method.
(iv) Term generation benefits from the equivalence reduction, avoiding duplicate work for equivalent terms.

$$\mathcal{V} = \{\, [\,] \quad \mathsf{list}\ T,\ \ :: \quad T \to \mathsf{list}\ T \to \mathsf{list}\ T,\ \ ++ \quad \mathsf{list}\ T \to \mathsf{list}\ T \to \mathsf{list}\ T,\ \ \mathrm{filter} \quad (T \to \mathsf{bool}) \to \mathsf{list}\ T \to \mathsf{list}\ T \,\}$$

$$\mathcal{C} = \{\, [\,], :: \,\}$$

$$\mathcal{E} = \{\, [\,] ++ l = l,\ \ (x :: xs) ++ l = x :: (xs ++ l),\ \ \mathrm{filter}\ p\ [\,] = [\,],\ \ \mathrm{filter}\ p\ (x :: xs) = \mathrm{if}\ p\,x\ \mathrm{then}\ x :: \mathrm{filter}\ p\ xs\ \mathrm{else}\ \mathrm{filter}\ p\ xs \,\}$$

Fig. 2. An example input to TheSy.

Running Example. To illustrate TheSy's theory exploration procedure, we introduce a simple running example based on a list ADT. The input given to TheSy is shown in Fig. 2; it consists of a vocabulary $\mathcal{V}$ (of which $\mathcal{C}$ is a subset of ADT constructors) and a set of known equalities $\mathcal{E}$. The vocabulary $\mathcal{V}$ contains the canonical list constructors [] and ::, and two basic list operations ++ (concatenate) and filter. The equalities $\mathcal{E}$ consist of the definitions of the latter two.

At a very high level, the following process is about to take place: TheSy generates symbolic terms representing length-bound lists, *e.g.*, [], $[v_1]$, $[v_2, v_1]$. Then, it will evaluate all combinations of function applications, up to a small depth, using these symbolic terms as arguments. If these evaluations yield common values for all possible assignments, the two application terms yielding them are conjectured to be equal. Since the evaluated expressions contain symbolic values, their result is a symbolic value. Comparing such symbolic values is done via congruence closure-based reasoning; we call this process *symbolic observational equivalence*, by way of analogy to observational equivalence [2] that is carried out using concrete values.

Out of the conjectures computed using symbolic observational equivalence, TheSy selects minimal ones according to a combined metric of compactness and generality. These are passed to a prover that employs both congruence closure and induction to verify the correction of the lemmas for *all* possible list values.

Some lemmas that TheSy can discover this way are:

$$\mathrm{filter}\ p\ (\mathrm{filter}\ p\ l) = \mathrm{filter}\ p\ l \qquad l_1 ++ (l_2 ++ l_3) = (l_1 ++ l_2) ++ l_3$$
$$\mathrm{filter}\ p\ l_1 ++ \mathrm{filter}\ p\ l_2 = \mathrm{filter}\ p\ (l_1 ++ l_2)$$

As briefly mentioned, our system design relies on congruence closure-based reasoning over universally quantified first-order formulas with uninterpreted functions. Congruence closure is weak but fast and constitutes one of the core procedures in SMT solvers [31,32]. On top of that, universally-quantified assumptions [4] are handled by formulating them as rewrite rules and applying some depth-bounded term rewriting as described in Subsect. 3.1. Additionally, TheSy implements a simple case splitting mechanism that enables reasoning on conditional expressions. Notably, this procedure *cannot* reason about recursive definitions since such reasoning routinely requires the use of induction. To that end, TheSy is geared towards discovering lemmas that can be proven by induction; a lemma is considered useful if it cannot be proven from existing lemmas by

congruence closure alone, that is, without induction. Discovering such lemmas and adding them to the background knowledge evidently increases the reasoning power of the prover, since at least the fact of their own validity becomes provable, which it was not before.

3 Preliminaries

This work relies heavily on term rewriting techniques, which is employed across multiple phases of the exploration. Term rewriting is implemented efficiently using equality graphs (e-graphs). In this section, we present some minimal background of both, which will be relevant for the exploration procedure described later.

3.1 Term Rewriting Systems

Consider a formal language $\mathcal{L}$ of terms over some vocabulary of symbols. We use the notation $\mathcal{R} = t_1 \dot{\rightarrow} t_2$ to denote a rewrite rule from t_1 to t_2. For a (universally quantified) semantic equality law $t_1 = t_2$, we would normally create *both* $t_1 \dot{\rightarrow} t_2$ and $t_2 \dot{\rightarrow} t_1$. We refrain from assigning a direction to equalities since we do not wish to restrict the procedure to strongly normalizing systems, as is traditionally done in frameworks based on the Knuth-Bendix algorithm [24]. Instead, we define equivalence when a sequence of rewrites can identify the terms in either direction. A small caveat involves situations where $\mathrm{FV}(t_1) \neq \mathrm{FV}(t_2)$, that is, one side of the equality contains variables that do not occur on the other. We choose to admit only rules $t_i \dot{\rightarrow} t_j$ where $\mathrm{FV}(t_i) \supseteq \mathrm{FV}(t_j)$, because when $\mathrm{FV}(t_i) \subset \mathrm{FV}(t_j)$, applying the rewrite would have to create new symbols for the unassigned variables in t_j, which results in a large growth in the number of symbols and typically makes rewrites much slower as a result.

This slight asymmetry is what motivates the following definitions.

Definition 1. *Given a rewrite rule $\mathcal{R} = t_1 \dot{\rightarrow} t_2$, we define a corresponding relation $\xrightarrow{\mathcal{R}}$ such that $s_1 \xrightarrow{\mathcal{R}} s_2 \iff s_1 = C[t_1\sigma] \wedge s_2 = C[t_2\sigma]$ for some context C and substitution σ for the free variables of t_1, t_2. (A* context *is a term with a single hole, and $C[t]$ denotes the term obtained by filling the hole with t.)*

Definition 2. *Given a relation $\xrightarrow{\mathcal{R}}$ we define its symmetric closure:*

$$t_1 \xleftrightarrow{\mathcal{R}} t_2 \iff t_1 \xrightarrow{\mathcal{R}} t_2 \ \vee \ t_2 \xrightarrow{\mathcal{R}} t_1$$

Definition 3. *Given a set of rewrite rules $G_{\mathcal{R}} = \{\mathcal{R}_i\}$, we define a relation as union of the relations of the rewrites: $\xleftrightarrow{\{\mathcal{R}_i\}} \ \widehat{=} \ \bigcup_i \xleftrightarrow{\mathcal{R}_i}$.*
In the sequel, we will mostly use its reflexive transitive closure, $\xleftrightarrow{\{\mathcal{R}_i\}}^*$.

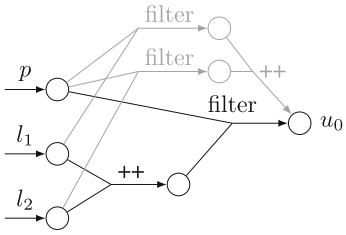

Fig. 3. An e-graph representing the expression filter p $(l_1 \mathbin{+\!\!+} l_2)$ (dark) and the equivalent expression filter p $l_1 \mathbin{+\!\!+}$ filter p l_2 (light).

The relation $\overset{\{\mathcal{R}_i\}}{\longleftrightarrow}^*$ is reflexive, transitive, and symmetric, so it is an equivalence relation over $\mathcal{L}$. Under the assumption that all rewrite rules in $\{\mathcal{R}_i\}$ are semantics preserving, for any equivalence class $[t] \in \mathcal{L}/\overset{\{\mathcal{R}_i\}}{\longleftrightarrow}^*$, all terms belonging to $[t]$ are definitionally equal. However, since $\mathcal{L}$ may be infinite, it is essentially impossible to compute $\overset{\{\mathcal{R}_i\}}{\longleftrightarrow}^*$. Any algorithm can only explore a finite subset $\mathcal{T} \subseteq \mathcal{L}$, and in turn, construct a subset of $\overset{\{\mathcal{R}_i\}}{\longleftrightarrow}^*$.

3.2 Compact Representation Using Equality Graphs

In order to be able to cover a large set of terms $\mathcal{T}$, we need a compact data structure that can efficiently represent many terms. Normally, terms are represented by their ASTs (Abstract Syntax Trees), but as there would be many instances of common subterms among the terms of $\mathcal{T}$, this would be highly inefficient. Instead, we adopt the concept of equality graphs (e-graphs) from automated theorem proving [15], which also saw uses in compiler optimizations and program synthesis [30,34,41], in which context they are known as Program Expression Graphs (PEGs). An e-graph is essentially a hypergraph where each vertex represents a set of equivalent terms (programs), and labeled, directed hyperedges represent function applications. Hyperedges therefore have exactly one target and zero or more sources, which form an ordered multiset (a vector, basically). Just to illustrate, the expression filter p $(l_1 \mathbin{+\!\!+} l_2)$ will be represented by the nodes and edges shown in dark in Fig. 3. The nullary edges represent the constant symbols (p, l_1, l_2), and the node u_0 represents the entire term. The expression filter p $l_1 \mathbin{+\!\!+}$ filter p l_2, which is equivalent, is represented by the light nodes and edges, and the equivalence is captured by sharing of the node u_0.

When used in combination with a rewrite system $\{\mathcal{R}_i\}$, each rewrite rule is represented as a premise pattern P and a conclusion pattern C. Applying a rewrite rule is then reduced to searching the e-graph for the search pattern and obtaining a substitution σ for the free variables of P. The result term is then obtained by substituting the free variables of C using σ. This term is added to the same equivalence class as the matched term (*i.e.* $P\sigma$), meaning they will both have the same root node. Consequently, a single node can represent a set

of terms exponentially large in the number of edges, all of which will always be equivalent modulo $\overset{\{\mathcal{R}_i\}}{\longleftrightarrow}^*$.

In addition, since hyperedges always represent functions, a situation may arise in which two vertices represent the same term: This happens if two edges $\bar{u} \xrightarrow{f} v_1$ and $\bar{u} \xrightarrow{f} v_2$ are introduced by $\{\mathcal{R}_i\}$ for $v_1 \neq v_2$. In a purely functional setting, this means that v_1 and v_2 are equal. Therefore, when such duplication is found, it is beneficial to *merge* v_1 and v_2, eliminating the duplicate hyperedge. The e-graph data structure therefore supports a vertex merge operation and a congruence closure-based transformation [44] that finds vertices eligible for merge to keep the overall graph size small. This procedure can be quite expensive, so it is only run periodically.

4 Theory Synthesis

In this section, we go into a more detailed description of the phases of theory synthesis and explain how they are combined within an iterative deepening loop. To simplify the presentation, we describe all the phases first, then explain how the output from the last phase is fed back to the next iteration to complete a feedback loop. We continue with the input from the running example in Sect. 2 (Fig. 2) and dive deeper by showing intermediate states encountered during the execution of TheSy on this input. Throughout the execution, TheSy maintains a state, consisting of the following elements:

- $\mathcal{V}$, a sorted vocabulary
- $\mathcal{C} \subseteq \mathcal{V}$, a subset of constructors for some or all of the types
- $\mathcal{E}$, a set of equations initially consisting only of the definitions of the (non-constructor) functions in $\mathcal{V}$
- $\mathcal{T}$, a set of terms, initially containing just atomic terms corresponding to symbols from $\mathcal{V}$.

4.1 Term Generation

The first step is to generate a set of terms over the vocabulary $\mathcal{V}$. For the purpose of generating universally-quantified conjectures, we introduce a set of uninterpreted symbols, which we will call *placeholders*. Let $\mathcal{T}_\mathcal{Y}$ be the set of types occurring as the type of some argument of a function symbol in $\mathcal{V}$. For each type τ occurring in $\mathcal{V}$ we generate placeholders $\overset{\tau}{\circ}_i$, two for each type (we will explain later why two are enough). These placeholders, together with all the symbols in $\mathcal{V}$, constitute the terms at depth 0.

At every iteration of deepening, TheSy uses the set of terms generated so far, and the (non-nullary) symbols of $\mathcal{V}$, to form new terms by placing existing ones in argument positions. For example, with the definitions from Fig. 2, we will have terms such as these at depths 1 and 2:

$$\begin{array}{ll} 1 & \text{filter } \overset{T\to\text{bool}}{\circ_1}\ \overset{\text{list } T}{\circ_1} \qquad\qquad \overset{\text{list } T}{\circ_1} \texttt{ ++ } \overset{\text{list } T}{\circ_2} \\ \hline 2 & [\,] \texttt{ ++ } \text{filter } \overset{T\to\text{bool}}{\circ_1}\ \overset{\text{list } T}{\circ_1} \qquad \overset{\text{list } T}{\circ_1} \texttt{ ++ } (\text{filter } \overset{T\to\text{bool}}{\circ_1}\ \overset{\text{list } T}{\circ_2}) \\ & \text{filter } \overset{T\to\text{bool}}{\circ_1}\ (\overset{\text{list } T}{\circ_1} \texttt{ ++ } \overset{\text{list } T}{\circ_2}) \qquad (\text{filter } \overset{T\to\text{bool}}{\circ_1}\ \overset{\text{list } T}{\circ_1}) \texttt{ ++ } (\text{filter } \overset{T\to\text{bool}}{\circ_1}\ \overset{\text{list } T}{\circ_2}) \end{array} \quad (1)$$

It is easy to see that $\mathsf{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ {}^{\mathsf{list}\ T}\circ_1$ and $[\,]\mathbin{+\!\!+}\mathsf{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ {}^{\mathsf{list}\ T}\circ_1$ are equivalent in any context; this follows directly from the definition of ++, available as part of $\mathcal{E}$. It is therefore acceptable to discard one of them without affecting completeness. TheSy does not discard terms—since they are merged in the e-graph, there is no need to—rather, it chooses the smaller term as representative when it needs one. This sort of *equivalence reduction* is present, in some way or another, in many automated reasoning and synthesis tools.

To formalize the procedure of generating and comparing the terms, in an attempt to discover new equality conjectures, we introduce the concept of *Syntax Guided Enumeration* (SyGuE). SyGuE is similar to Syntax Guided Synthesis (SyGuS for short [3]) in that they both use a formal definition of a language to find program terms solving a problem. They differ in the problem definition: while SyGuS is defined as a search for a correct program over the well-formed programs in the language, SyGuE is the sub-problem of iterating over *all distinct* programs in the language. SyGuS solvers may be improved using a smart search algorithm, while SyGuE solvers need an efficient way to eliminate duplicate terms, which may depend on the definition of program equivalence. We implement our variant of SyGuE, over the equivalence relation $\xleftrightarrow{\{\mathcal{R}_i\}^*}$, using the aforementioned e-graph: by applying and re-applying rewrite rules, provably equivalent terms are naturally *merged* into hyper-vertices, representing equivalence classes.

4.2 Conjecture Inference and Screening

Of course, in order to discover *new* conjectures, we cannot rely solely on term rewriting based on $\mathcal{E}$. To find more equivalent terms, TheSy carries on to generate a second set of terms, called *symbolic examples*, this time using only the constructors $\mathcal{C} \subset \mathcal{V}$ and uninterpreted symbols for leaves. This set is denoted $\mathcal{S}^\tau$, where τ is an algebraic datatype participating in $\mathcal{V}$ (if several such datatypes are present, one $\mathcal{S}^\tau$ per type is constructed). The depth of the symbolic examples (i.e. depth of applied constructors) is also bounded, but it is independent of the current term depth and does not increase during execution. For example, using the constructors of $\mathsf{list}\ T$ with an example depth of 2, we obtain the symbolic examples $\mathcal{S}^{\mathsf{list}\ T} = \{[], v_1 :: [], v_2 :: v_1 :: []\}$, corresponding to lists of length up to 2 having arbitrary element values. Intuitively, if two terms are equivalent for all possible assignments of symbolic examples to ${}^{\mathsf{list}\ T}\circ_i$, then we are going *hypothesize* that they are equivalent for all list values. This process is very similar to observational equivalence as used by program synthesis tools [2,42], but since it uses the symbolic value terms instead of concrete values, we dub it *symbolic observational equivalence* (SOE).

Consider, for example, the simple terms ${}^{\mathsf{list}\ T}\circ_1$ and ${}^{\mathsf{list}\ T}\circ_1 \mathbin{+\!\!+} [\,]$. In placeholder form, none of the rewrite rules derived from $\mathcal{E}$ applies, so it cannot be determined that these terms are, in fact, equivalent. However, with the symbolic list examples above, the following rewrites are enabled:

$$[\,] \mathbin{+\!\!+} [\,] \overset{\{\mathcal{R}_i\}}{\longleftrightarrow}{}^{*} [\,] \qquad v_1 :: [\,] \mathbin{+\!\!+} [\,] \overset{\{\mathcal{R}_i\}}{\longleftrightarrow}{}^{*} v_1 :: [\,] \qquad v_2 :: v_1 :: [\,] \mathbin{+\!\!+} [\,] \overset{\{\mathcal{R}_i\}}{\longleftrightarrow}{}^{*} v_2 :: v_1 :: [\,]$$

A similar case can be made for the two bottom terms in (1). For symbolic values $l_1, l_2 \in \mathcal{S}^{\mathsf{list}\,T}$, it can be shown that

$$\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ (l_1 \mathbin{+\!\!+} l_2) \overset{\{\mathcal{R}_i\}}{\longleftrightarrow}{}^{*} (\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ l_1) \mathbin{+\!\!+} (\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ l_2)$$

In fact, it is sufficient to substitute for ${}^{\mathsf{list}\,T}\circ_1$, while *leaving* ${}^{\mathsf{list}\,T}\circ_2$ *alone, uninterpreted*: e.g., $\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ ([\,] \mathbin{+\!\!+} {}^{\mathsf{list}\,T}\circ_2) \overset{\{\mathcal{R}_i\}}{\longleftrightarrow}{}^{*} (\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ [\,]) \mathbin{+\!\!+} (\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ {}^{\mathsf{list}\,T}\circ_2)$. This reduces the number of equivalence checks significantly, and is more than a mere heuristic: since we are going to rely on a prover that proceeds by applying induction to one of the arguments, it makes perfect sense to only bound that argument. If computation is blocked on the second argument, we would prefer to first infer an auxiliary lemma first, then use it to discover the blocked lemma later. See Example 1 below for an idea of when this situation arises.

The attentive reader may notice that the cases of $v_1 :: [\,]$ and $v_2 :: v_1 :: [\,]$ are a bit more involved: to proceed with the rewrite of filter, the expressions ${}^{T\to\mathsf{bool}}\circ_1\ v_1$, ${}^{T\to\mathsf{bool}}\circ_1\ v_2$ must be resolved to either *true* or *false*. However, the predicate ${}^{T\to\mathsf{bool}}\circ_1$ as well as the arguments $v_{1,2}$ are uninterpreted. In this case, TheSy is required to perform a *case split* in order to enable the rewrites and unify the symbolic terms separately in each of the resulting four (2^2) cases. Notice that leaving ${}^{T\to\mathsf{bool}}\circ_1$ uninterpreted means that the cases are only split when evaluation is blocked by one or more rewrite rule applications, potentially saving some branching. The following steps are then carried out for each case.

TheSy applies all the available rewrite rules to the entire e-graph, containing all the terms and symbolic examples. For every two terms t_1, t_2 such that for all viable substitutions σ of placeholders to symbolic examples of the corresponding types, $t_1\sigma$ and $t_2\sigma$ were shown equal—that is, ended up in the same equivalence class of the e-graph—the conjecture $t_1 \overset{?}{=} t_2$ is emitted. *E.g.*, in the case of the running example:

$$\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ ({}^{\mathsf{list}\,T}\circ_1 \mathbin{+\!\!+} {}^{\mathsf{list}\,T}\circ_2) \overset{?}{=} (\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ {}^{\mathsf{list}\,T}\circ_1) \mathbin{+\!\!+} (\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ {}^{\mathsf{list}\,T}\circ_2)$$

In the presence of multiple cases, the results are intersected, so that a conjecture is emitted only if it follows from all the cases.

Screening. Generating all the pairs according to the above criteria potentially creates many "obvious" equalities, which are valid propositions, but do not contribute to the overall knowledge and just clutter the prover's state. For example,

$$\mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ ({}^{\mathsf{list}\,T}\circ_1 \mathbin{+\!\!+} {}^{\mathsf{list}\,T}\circ_2) \overset{?}{=} \mathrm{filter}\ {}^{T\to\mathsf{bool}}\circ_1\ ({}^{\mathsf{list}\,T}\circ_1 \mathbin{+\!\!+} ([\,] \mathbin{+\!\!+} {}^{\mathsf{list}\,T}\circ_2))$$

which follows from the definition of ++ and has nothing to do with filter. The synthesizer avoids generating such candidates, by choosing at most one term from every equivalence class of placeholder-form terms induced during the term

generation phase. If both sides of the equality conjecture belong to the same equivalence class, the conjecture is dropped altogether.

The conjectures that remain are those equalities $t_1 \stackrel{?}{=} t_2$ where t_1 and t_2 got merged for all the assignments $\mathcal{S}^\tau$ to some $\overset{\tau}{\circ}_1$, and, furthermore, t_1 and t_2 themselves *were not* merged in placeholder form, prior to substitution. Such conjectures, if true, are guaranteed to increase the knowledge represented by $\mathcal{E}$ as (at least) the equality $t_1 = t_2$ was not previously provable using term rewriting and congruence closure.

4.3 Induction Prover

For practical reasons, the prover employs the following induction tactic:

- *Structural* induction based on the provided constructors ($\mathcal{C}$).
- The *first* placeholder of the inductive type is selected as the decreasing argument.
- Exactly *one* level of induction is attempted for each candidate.

The reasoning behind this design choice is that for every multi-variable term, *e.g.* $\overset{\mathsf{list}\ T}{\circ}_1 \texttt{++} \overset{\mathsf{list}\ T}{\circ}_2$, the synthesizer also generates the symmetric counterpart $\overset{\mathsf{list}\ T}{\circ}_2 \texttt{++} \overset{\mathsf{list}\ T}{\circ}_1$. So electing to perform induction on $\overset{\mathsf{list}\ T}{\circ}_1$ does not impede generality.

In addition, if more than one level of induction is needed, the proof can (almost) always be revised by factoring out the inner induction as an auxiliary lemma. Since the synthesizer produces *all* candidate equalities, that inner lemma will also be discovered and proved with one level of induction. Lemmas so proven are added to $\mathcal{E}$ and are available to the prover, so that multiple passes over the candidates can gradually grow the set of provable equalities.

When starting a proof, the prover never needs to look at the base case, as this case has already been checked during conjecture inference. Recall that placeholders $\overset{\tau}{\circ}_1$ are instantiated with bounded-depth expressions using the constructors of τ, and these include all base cases (non-recursive constructors) by default. For the example discussed above, the case of $\mathsf{filter}\ \overset{T\to\mathsf{bool}}{\circ}_1\ ([]\ \texttt{++}\ \overset{\mathsf{list}\ T}{\circ}_2) = (\mathsf{filter}\ \overset{T\to\mathsf{bool}}{\circ}_1\ []) \texttt{++} (\mathsf{filter}\ \overset{T\to\mathsf{bool}}{\circ}_1\ \overset{\mathsf{list}\ T}{\circ}_2)$ has been discharged early on, otherwise the conjecture would not have come to pass. The prover then turns to the induction step, which is pretty routine but is included in Fig. 4 for completeness of the presentation.

It is worth noting that the conjecture inference, screening and induction phases utilize a common reasoning core based on rewriting and congruence closure. In situations where the definitions include conditions such as $\mathsf{match}\ p\ x$ in Fig. 4 (in this case, desugared from $\mathsf{if}\ p\ x$), the prover also performs automatic case split and distributes equalities over the branches. Details and specific optimizations are described in Sect. 5.

$$
\begin{array}{lll}
\textit{Assume} & \text{filter } p\ (xs +\!\!+ l_1) = \text{filter } p\ xs +\!\!+ \text{filter } p\ l_1 & \\
\textit{Prove} & \text{filter } p\ ((x :: xs) +\!\!+ l_1) = \text{filter } p\ (x :: xs) +\!\!+ \text{filter } p\ l_1 & \\
\textit{via} & (1)\ \ \text{filter } p\ ((x :: \text{xs}) +\!\!+ l_1) = \text{filter } p\ (x ::(\text{xs} +\!\!+ l_1)) & \\
 & (2) \qquad = \text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: \text{filter } p\ (xs +\!\!+ l_1) & \\
 & \qquad\qquad\qquad\qquad\quad \textit{false} \Rightarrow \text{filter } p\ (xs +\!\!+ l_1) & \\
(\textit{IH}) & (3) \qquad = \text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: (\text{filter } p\ xs +\!\!+ \text{filter } p\ l_1) & \\
 & \qquad\qquad\qquad\qquad\quad \textit{false} \Rightarrow \text{filter } p\ xs +\!\!+ \text{filter } p\ l_1 & \\
 & (4)\ \ \text{filter } p\ (x :: xs) +\!\!+ \text{filter } p\ l_1 & \\
 & \qquad\quad = \big(\text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: \text{filter } p\ xs & \\
 & \qquad\qquad\qquad\qquad\quad \textit{false} \Rightarrow \text{filter } p\ xs\big) +\!\!+ \text{filter } p\ l_1 & \\
 & (5) \qquad = \text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: (\text{filter } p\ xs +\!\!+ \text{filter } p\ l_1) & \\
\square & \qquad\qquad\qquad\qquad\quad \textit{false} \Rightarrow \text{filter } p\ xs +\!\!+ \text{filter } p\ l_1 &
\end{array}
$$

Fig. 4. Example proof by induction based on congruence closure and case splitting.

Speculative Generalization. When the prover receives a conjecture with multiple occurrences of a placeholder, *e.g.* $\circ_1^{\text{list } T} +\!\!+ (\circ_2^{\text{list } T} +\!\!+ \circ_1^{\text{list } T}) \stackrel{?}{=} (\circ_1^{\text{list } T} +\!\!+ \circ_2^{\text{list } T}) +\!\!+ \circ_1^{\text{list } T}$, it is designed to first speculate a more general form for it by replacing the multiple occurrences with fresh placeholders. Recall that in Subsect. 4.1 we argued that two placeholders of each type is going to be sufficient; this is the mechanism that enables it. There is more than one way to generalize a given conjecture: for this example, there are two ways (up to alpha-renaming):

$$\circ_1^{\text{list } T} +\!\!+ (\circ_2^{\text{list } T} +\!\!+ \circ_3^{\text{list } T}) \stackrel{?}{=} (\circ_1^{\text{list } T} +\!\!+ \circ_2^{\text{list } T}) +\!\!+ \circ_3^{\text{list } T} \qquad \circ_1^{\text{list } T} +\!\!+ (\circ_2^{\text{list } T} +\!\!+ \circ_3^{\text{list } T}) \stackrel{?}{=} (\circ_3^{\text{list } T} +\!\!+ \circ_2^{\text{list } T}) +\!\!+ \circ_1^{\text{list } T}$$

The prover must attempt both. Failing that, it would fall back to the original conjecture. Formally, given an equality conjecture $s = t$ we can consider an assignment σ such that $r = s\sigma, q = t\sigma$; where the original conjecture uses an assignment with only two values per type. The prover thus must iterate through different assignments σ_i with more possible values per type, and attempt to prove a new conjecture $r\sigma_i = q\sigma_i$. This incurs more work for the prover but is well worth its cost compared to a-priori generation of terms with three placeholders.

4.4 Looping Back

The equations obtained from Subsect. 4.3 are fed back in four different but interrelated ways. The first, inner feedback loop is from the induction prover to itself: the system will attempt to prove the smaller lemmas first, so that when proving the larger ones, these will already be available as part of $\mathcal{E}$. This enables more proofs to go through. The second feedback loop uses the lemmas obtained to filter out proofs that are no longer needed. The third, outer loop is more interesting: as equalities are made into rewrite rules, additional equations may

now pass the inference phase, since the symbolic evaluation core can equate more terms based on this additional knowledge. The fourth resonates with the third, applying the new rewrite rules acts as an equality reduction mechanism, reducing the number of hyperedges added to the e-graph during term generation.

It is worth noting that while concrete observational equivalence uses a trivially simple equivalence checking mechanism with the trade-off that it may generate many incorrect equalities, our *symbolic* observational equivalence is conservative in the sense that a symbolic value may represent infinitely many concrete inputs, and only if the synthesizer can *prove* that two terms will evaluate to equal values on *all* of them, by way of constructing a small proof, are they marked as equivalent. This means that some actually-equivalent terms may be "blocked" by the inference phase, which cannot happen when using concrete values—but also means that having additional inference rules ($\mathcal{E}$) can improve this equivalence checking, potentially leading to more discovered lemmas. This property of TheSy is appealing because it allows an explored theory to evolve from basic lemmas to more complex ones.

Example 1 (Lemma seeding). To understand this last point, consider the standard definition of list reversal for the list datatype:

$$\begin{aligned} \text{rev}\ [\,] &= [\,] \\ \text{rev}\ (x :: xs) &= \text{rev}\ xs \mathbin{+\!\!+} (x :: [\,]) \end{aligned}$$

Given the terms $t_1 = \text{rev}\ (\circ_1^{\text{list}\,T} \mathbin{+\!\!+} \circ_2^{\text{list}\,T})$ and $t_2 = \text{rev}\ \circ_2^{\text{list}\,T} \mathbin{+\!\!+} \text{rev}\ \circ_1^{\text{list}\,T}$, symbolic observational equivalence with the assignments $\{\circ_1^{\text{list}\,T} \mapsto \mathcal{S}^{\text{list}\,T}\}$ fails to unify them. This is due to ++ being defined by induction on its first argument, hence, *e.g.*—

$$\begin{aligned} \text{rev}\ (v_2 :: v_1 :: [\,] \mathbin{+\!\!+} \circ_2^{\text{list}\,T}) \quad &\to^* \quad (\text{rev}\ \circ_2^{\text{list}\,T} \mathbin{+\!\!+} (v_1 :: [\,])) \mathbin{+\!\!+} (v_2 :: [\,]) \\ \text{rev}\ \circ_2^{\text{list}\,T} \mathbin{+\!\!+} \text{rev}\ v_2 :: v_1 :: [\,] \quad &\to^* \quad \text{rev}\ \circ_2^{\text{list}\,T} \mathbin{+\!\!+} (v_1 :: v_2 :: [\,]) \end{aligned}$$

Without the associativity property of ++, it would not be possible to show that these symbolic values are equivalent, so the conjecture $t_1 \stackrel{?}{=} t_2$ will not even be generated. Luckily, having proven $\circ_1^{\text{list}\,T} \mathbin{+\!\!+} (\circ_2^{\text{list}\,T} \mathbin{+\!\!+} \circ_3^{\text{list}\,T}) \stackrel{?}{=} (\circ_1^{\text{list}\,T} \mathbin{+\!\!+} \circ_2^{\text{list}\,T}) \mathbin{+\!\!+} \circ_3^{\text{list}\,T}$, these rewrites are "unblocked", so that the equality can be conjectured and ultimately proven.

One caveat is that whenever $\mathcal{E}$ is updated by the addition of a new lemma, some of the previously emitted conjectures may consequently become redundant. Moreover, conjectures that were passed to the prover before but failed validation may now succeed, and new ones may be emitted in the generation phase. To take these into account, the actual loop performed by TheSy is a bit more involved than has been described so far. For each term depth, TheSy performs all phases as described, but each time a lemma is discovered TheSy re-runs the conjecture generation, screening, and prover phases. Only when no more conjectures are available does TheSy increase the term depth and generate new terms.

5 Evaluation

We implemented TheSy in Rust, using the e-graph manipulation library *egg* [44]. TheSy accepts definitions in SMTLIB-2.6 format [6], based on the UF theory (uninterpreted functions), limited to universal quantifications. Type declarations occurring in the input are collected and comprise $\mathcal{V}$; universal equalities form $\mathcal{E}$ and are translated into rewrite rules (either uni- or bidirectional, as explained in Subsect. 3.1). Then SyGuE is performed on $\mathcal{V}$, generating candidate conjectures using SOE. SyGuE uses *egg* for equivalence reduction, and SOE uses it for comparing symbolic values. Conjectures are then dismissed using TheSy's induction-based prover. This is done in an iterative deepening loop.

Case Split. Both SOE and the prover use a case splitting mechanism; This mechanism detects when rewriting cannot match due to an opaque value (an uninterpreted symbol), and applies case splitting according to the constructors of relevant ADTs. However, doing so for every rule is too costly and, in most cases, redundant—TheSy generates a variety of terms, so if one term is blocked due to an uninterpreted symbol, another one exists with a symbolic example instead. A situation where this is *not* the case is when *multiple* uninterpreted symbols block the rewrite (recall that TheSy only substitutes one placeholder per term with symbolic examples). To illustrate, consider the case in Fig. 4 where both the list $x :: xs$ and $p\,x$ are used in match expressions, therefore a case split is needed by $p\,x \in \{\mathit{true}, \mathit{false}\}$. Therefore, TheSy only performs case splitting for rewrite rules that require multiple match patterns but only one is blocked.

The splitting mechanism itself, operates by copying the e-graph and applying the term rewriting logic separately for each case. Each copy then yields a partition of the existing equivalence classes. These partitions are intersected between all cases, and each of the resulting intersections lead to merging of equivalence classes in the original e-graph. It is worth noting that TheSy never needs to backtrack a case split it has elected to apply. As a consequence, execution time is not exponential in the total number of case splits performed, only in the nesting level of such splits (which is bounded by 2 in our experiments).

We compare TheSy to the most recent and closely related theory exploration system, Hipster [23]—which is based on random testing (backed by QuickSpec [38]) with proof automations from and frontend in Isabelle/HOL [33]. Hipster represents the culmination of several works on existing theory exploration (see Sect. 6). Both systems generate a set of proved lemmas as output, each such set encompassing a conceptual volume of knowledge that was discovered automatically. We note that the same knowledge can be represented in various ways, so directly comparing the sets of lemmas is going to be meaningless.

5.1 Evaluating Theory Exploration Quality

We define a comparison method for two theory exploration systems A and B starting from a common initial theory (defined as a set of closed formulas) $\mathcal{T}$. As a metric for the quality and efficacy of results obtained from theory exploration, and, therefore, their perceived usefulness, we use the notion of *knowledge*

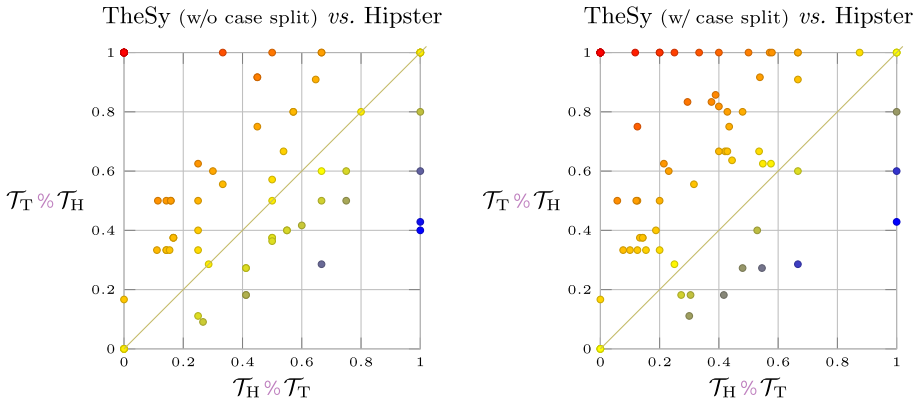

Fig. 5. A scatter plot showing the ratio of lemmas in theories discovered by each tool that were subsumed by the theory discovered by its counterpart (T = TheSy, H = Hipster). Each point represents a single test case. The vertical axis shows how many of the lemmas discovered by Hipster were subsumed by those discovered by TheSy, and the horizontal axis shows the converse.

(inspired by "knowledge base" in Theorema [8]). A theory $\mathcal{T}$ in a given logical proof system induces a collection of attainable knowledge, $\mathcal{K}_\mathcal{T} = \{\varphi \mid \mathcal{T} \vdash \varphi\}$, that is, characterized by the set of (true) statements that can be proven based on $\mathcal{T}$. In practice, a "pure" notion of knowledge based on provability is impractical, because most interesting logics are undecidable, and automated proving techniques cannot feasibly find proofs for all true statements. We, therefore, parameterize knowledge relative to a *prover*—a procedure that always terminates and can prove a subset of true statements. Termination can be achieved by restricting the space of proofs by either size or resource bounds. We say that $\mathcal{T} \overset{S}{\vdash} \varphi$ when a prover, S, is able to verify the validity of φ in a theory $\mathcal{T}$. A more realistic characterization of knowledge would then be $\mathcal{K}^S_\mathcal{T} = \{\varphi \mid \mathcal{T} \overset{S}{\vdash} \varphi\}$. Assuming that the prover S is fixed, a theory $\mathcal{T}'$ is said to *increase knowledge* over $\mathcal{T}$ when $\mathcal{K}^S_{\mathcal{T}'} \supset \mathcal{K}^S_\mathcal{T}$.

We utilize the notion of $\mathcal{K}^S_\mathcal{T}$ described above to test the knowledge gained by A against that of B, and vice versa. We take the set of lemmas $\mathcal{T}_A$ generated by A and check whether it is subsumed by $\mathcal{T}_B$, generated by B, by checking whether $\mathcal{T}_A \subseteq \mathcal{K}^S_{\mathcal{T} \cup \mathcal{T}_B}$; we then carry out the same comparison with the roles of A and B reversed. A working assumption is that both A and B include some mechanism for screening redundant conjectures. That is, a component that receives the current set of known lemmas T_i and a conjecture φ and decides whether the conjecture is redundant. It is important to choose S such that whenever A (or B) discards φ, due to redundancy, it holds that $\varphi \in \mathcal{K}^S_{T_i}$.

Incorporating the solver into the comparison makes the evaluation resistant to large amounts of trivial lemmas, as they will be discarded by A or B. It is

still possible for some lemmas to be "better" than others, so knowledge is not uniformly distributed; this is hard to quantify, though. A few possible measures of usefulness come to mind, such as lemma utilization in a task (such as proof search), proof complexity, or matching to a given context, but given just the exploration task, there is not sufficient information to apply them. A first approximation is to consider the discovered lemmas themselves, *i.e.*, $\mathcal{T}_A \cup \mathcal{T}_B$, as representing proof objectives. In doing so, we pit A and B in direct contest with one another. We choose this avenue because it is straightforward to apply, admitting that it may be inaccurate in some cases.

To evaluate our approach and its implementation, we run both TheSy and Hipster on functional definitions collected from the TIP 2015 benchmark suite [11], specifically the IsaPlanner [21] benchmarks (85 benchmarks in total), for compatibility between the two systems. TIP benchmarks also contain goal propositions, but for the purpose of evaluating the exploration technique, these are redacted. This experiment uses the simple rewrite-driven congruence-closure decision procedure with a case split mechanism in the role of the solver, S, occurring in the definition of knowledge $\mathcal{K}$. Hipster uses Isabelle/HOL's simplifier as a conjecture redundancy filtering mechanism, which is in itself a simple rewrite-driven decision procedure, therefore S provides a suitable comparison. We compute the portion of lemmas found by Hipster that were provable (by S) from TheSy's results and vice versa. In other words, we check the ratio given by $|\mathcal{T}_A \cap \mathcal{K}^S_{\mathcal{T}\cup\mathcal{T}_B}| \,/\, |\mathcal{T}_A|$, which we denote $\mathcal{T}_B \,\%\, \mathcal{T}_A$, in both directions. Figure 5 displays the ratios, where each point represents a single test case. Points above the diagonal line represent test cases where TheSy's ratio was higher and for points under the line Hipster's ratio was higher. We conduct this experiment twice: Once with the case-splitting mechanism of TheSy turned off for its exploration, and once with it turned on. (Hipster does not have such a switch as it always generates concrete values.) The reason for this is that case splitting increases the running time significantly (as we show next), so we want to evaluate its contribution to the discovery of lemmas. Comparing the two charts, while TheSy performs reasonably well compared to Hipster without case splitting (in 48 out of the 85 TheSy's ratio was better and equal in 12), enabling it leads to a clear advantage (in 65 out of the 85 TheSy's ratio was better and equal in 6).

Performance. To compare runtime efficiency, we consider the time it took to fully explore the IsaPlanner test suite. We consider an exploration "full" when it has finished enumerating all the terms, and associated candidate conjectures, up to the depth bound ($k = 2$)[2] with TheSy or size bound with Hipster ($s = 7$), and check them; or when a timeout of one hour is reached, whichever is sooner. We then sort the benchmarks from shortest- to longest-running for each of the tools, and report the accumulated time to explore the first i benchmarks ($i = 1..85$). The results are shown in the graph in Fig. 6, for Hipster, TheSy with case split disabled, and TheSy with case split enabled. In both configurations,

[2] Our experience shows that choosing larger ks greatly affects the run-time, but does not lead to many useful lemmas.

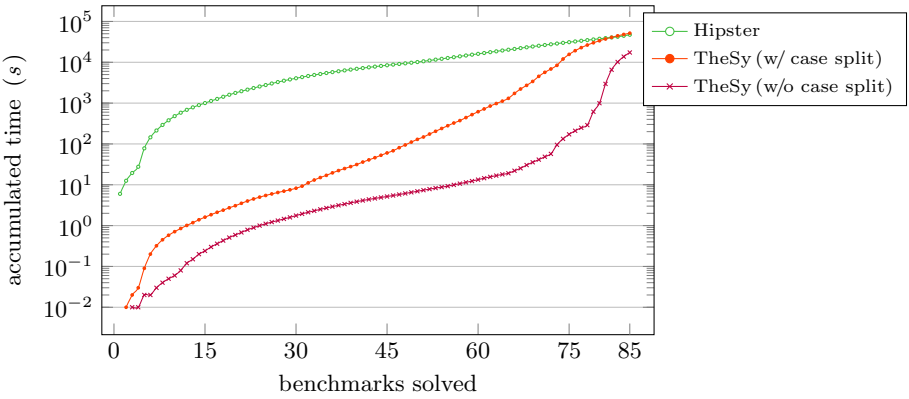

Fig. 6. Time to fully explore the 85 IsaPlanner benchmarks. A full exploration is considered one where either all terms up to the depth bound have been enumerated or a timeout of 1 h has been reached. The y axis shows the amount of time needed to complete the first x benchmarks, when they are sorted from shortest- to longest-running. (Time scale is logarithmic; lower is better.)

TheSy is very fast for the lower percentiles, but begins to slow down, due to case splitting, towards the end of the line. To illustrate, in the 25th percentile TheSy was ~380 times faster (0.48 s *vs.* 182.47 s); in the 50th percentile, ~57 times faster (5.28 s *vs.* 305.37 s); and in the 75th percentile, ~6 times faster (141.24 to 883.8). Overall TheSy took 51.6K seconds and Hipster 47.1K, meaning Hipster was ~1.1 times faster. It is evident from the chart that case splitting is largely responsible for the longer execution times. Without case splitting, TheSy is much faster, and completes all 85 benchmarks in less time than it takes Hipster. Of course, in that mode of operation, TheSy finds fewer lemmas (as shown in Fig. 5), but is still superior to Hipster. Future work needs to focus on improving the case-splitting mechanism, similar to their treatment in SAT and SMT, allowing TheSy to deal with such theories more efficiently.

5.2 Efficacy to Automated Proving

While the mission statement of TheSy is solely to provide lemmas based on core theories, we wish to claim that such discovered theories are beneficial toward proving theorems in general, based on the same core theory. We used a collection of benchmarks for induction proofs used by CVC4 [37], and conducted the following experiment: First, the proof goals are skipped and only the symbol declarations and provided axioms are used to construct an input to TheSy. Then, whenever a new lemma is discovered and passes through the prover, we also attempt to prove the goal—utilizing the same mechanism used for vetting conjectures. As soon as the latter goes through, the exploration process is aborted, and all lemmas collected are discarded. The experiments are thus independent across the individual benchmarks.

Table 1. Results of the CVC4 benchmark suite (number of successful proofs in each category).

	Total	Z3	CVC4	CVC4+ig	TheSy
clam	136	25	20	108	102
hipspec	42	6	7	33	29
isaplanner	87	35	34	79	47
leon	46	9	9	40	9
Total	311	75	70	260	187

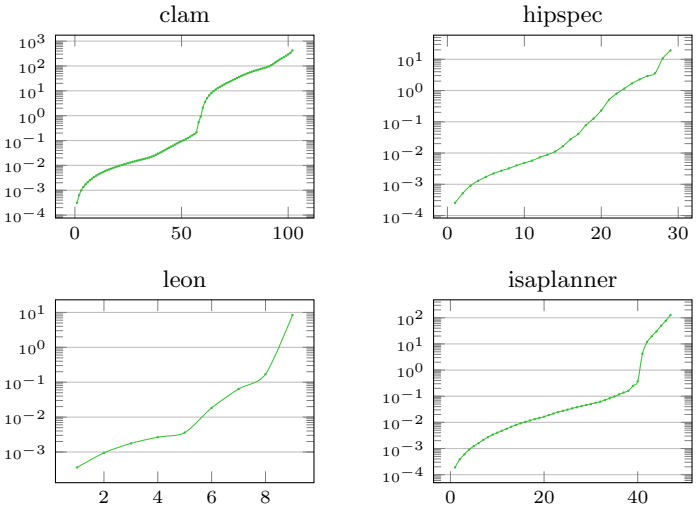

Fig. 7. Accumulated time-to-solve for each of the benchmark suites from the CVC4 collection. The y axis shows the amount of time needed to complete the first x (successful) proofs, when benchmarks are sorted from shortest- to longest-running.

Even though this setting is unfavorable to TheSy—because it does not take advantage of the fact that theory exploration can be done offline, then its results re-used for proofs over the same core theory—we report considerable success in solving these benchmarks. Out of the 311 benchmarks, our theory exploration + simple-minded induction was able to prove 187 (with a 5-min timeout, same as in the original CVC4 experiments). For comparison, Z3 and CVC4 (without conjecture generation) were able to prove 75 and 70 of them, respectively. This shows that the majority of instances were not solvable without the use of induction. CVC4 with its conjecture generation enabled was able to solve 260 of them. Table 1 shows the number of successful proofs achieved for each of the four suites. Figure 7 shows the accumulated time required for the benchmarks; the vast majority of the success cases occur early on, because in some cases a rather small auxiliary lemma is all that is needed to make the proof go through.

6 Related Work

Equality Graphs. Originally brought into use for automated theorem proving [15], e-graphs were popularized as a mechanism for implementing low-level compiler optimizations [41], under the name *PEGs*. These e-graphs can be used to represent a large program space compactly by packing together equivalent programs. In that sense they are similar to Version Space Algebras [26], but their prime objective is entirely different. While VSAs focus on efficient intersections, e-graphs are used to saturate a space of expressions with all equality relations that can be inferred. They have found use in optimizing expressions for more than just speed, for example to increase numerical stability of floating-point programs in Herbie [34]. There are two key differences in the way e-graphs are used in this work compared to prior: (i) equality laws are not hard-coded nor fixed, they are fertilized as the system proves more lemmas automatically; (ii) saturation cannot be guaranteed or even obtained in all cases, which we overcome by a bound on rewrite-rule application depth. (The latter point is an indirect consequence of the former.)

Automated Theorem Provers. Many systems rely on known theorems or are designed to support users in semi-automated proving. Congruence closure is also a proven method for tautology checking in automated theorem provers, such as Vampire [25], and is used as a decision procedure for reasoning about equality in leading SMT solvers Z3 [14] and CVC4 [5]. There, it is limited mostly to first-order reasoning, but can essentially be applied unchanged to higher-level scenarios such as ours.

Related to theory exploration, but using separate techniques, are Zipperposition [13], and the conjecture generation mechanism implemented as part of the induction prover in CVC4 [37]. It should be noted, that these are directed toward a specific proof goal, as opposed to theory exploration, which is presumed to be an offline phase. As such, the above two techniques incorporate generation of inductive hypotheses into the saturation proof search/SMT procedure, respectively.

Theory Exploration. IsaCoSy [22] pioneered the use of synthesis techniques for bottom-up lemma discovery. IsaCoSy combines equivalence reduction with counterexample-guided inductive synthesis (CEGIS [40]) for filtering candidate lemmas. This requires a solver capable of generating counterexamples to equivalence. Subsequent development was based on random generation of test values, as implemented in QuickSpec [38] for reasoning about Haskell programs, later combined with automated provers for checking the generated conjectures [10,20]. We have mentioned the deficiencies of using concrete values (as opposed to symbolic ones) and random testing in Sect. 1 and make an empirical comparison with Hipster, a descendent of IsaCoSy and QuickSpec, in Sect. 5.

Inductive Synthesis. In the area of SyGuS [3], tractable bottom-up enumeration is commonly achieved by some form of equivalence reduction [39]. When dealing with concrete input-output examples, observational equivalence [2,42] is very

effective. The use of symbolic examples in synthesis has been suggested [17], but to the best of our knowledge, ours is the only setting where symbolic observational equivalence has been applied. Inductive synthesis, in combination with abduction [16], has also been used to infer specifications [1], although not as an exploration method but as a supporting mechanism for verification.

7 Conclusion

We described a new method for theory exploration, which differentiates itself from existing work by basing the reasoning on a novel engine based on term rewriting. The new approach differs from previous work, specifically those based on testing techniques, in that:

1. This lightweight reasoning is purely symbolic, supporting value abstraction and performs better then prior art.
2. Functions are naturally treated as first-class objects, without specific support implementation.
3. The only needed input is the code defining the functions involved, and no support code such as a specific theory solver or random value generators.
4. TheSy has a unique feedback loop between the prover and the synthesizer, allowing more conjectures to be found and proofs to succeed.

By creating a feedback loop between the four different phases, term generation, conjecture inference, conjecture screening and induction prover, this system manages to efficiently explore many theories. This goes beyond similar feedback loops in existing tools, aiming to reduce false and duplicate conjectures. As explained in Subsect. 4.2, this form is also present in TheSy, but TheSy utilizes this feedback in more phases of the computation.

Theory exploration carries practical significance to many automated reasoning tasks, especially in formal methods, verification and optimization. Complex properties lead to an ever-growing number of definitions and associated lemmas, which constitute an integral part of proof construction. These lemmas can be used for SMT solving, automated and interactive theorem proving, and as a basis for equivalence reduction in enumerative synthesis. The term rewriting-based method that we presented in this paper is simple, highly flexible, and has already shown results surpassing existing exploration methods. The generated lemmas allow even this simple method to prove conjectures that normally require sophisticated SMT extensions. Our main conclusion is that deductive techniques and symbolic evaluation can greatly contribute to theory exploration, in addition to their existing applications in invariant and auxiliary conjecture inference.

Acknowledgements. This research was supported by the Israeli Science Foundation (ISF) Grants No. 243/19 and 2740/19 and by the United States-Israel Binational Science Foundation (BSF) Grant No. 2018675.

References

1. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 789–801. Association for Computing Machinery, New York (2016)
2. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_67
3. Alur, R., et al.: Syntax-guided synthesis. Dependable Softw. Syst. Eng. **40**, 1–25 (2015)
4. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 214–230. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_13
5. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
7. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the Leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA 2013. Association for Computing Machinery, New York (2013)
8. Buchberger, B.: Theory exploration with theorema. Analele Universitatii Din Timisoara, ser. Matematica-Informatica **38**(2), 9–32 (2000)
9. Buchberger, B., et al.: Theorema: towards computer-aided mathematical theory exploration. J. Appl. Logic **4**(4), 470–504 (2006)
10. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 392–406. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_27
11. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: TIP: tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 333–337. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_23
12. The Coq Development Team: The Coq Proof Assistant Reference Manual, version 8.7 (October 2017)
13. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 172–188. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_10
14. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
15. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
16. Dillig, I., Dillig, T., Li, B., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. Int. J. Softw. Tools Technol. Transf. **19**(5), 535–547 (2015). https://doi.org/10.1007/s10009-015-0397-7

17. Drachsler-Cohen, D., Shoham, S., Yahav, E.: Synthesis with abstract examples. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 254–278. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_13
18. Einarsdóttir, S.H., Johansson, M., Åman Pohjola, J.: Into the infinite - theory exploration for coinduction. In: Fleuriot, J., Wang, D., Calmet, J. (eds.) AISC 2018. LNCS (LNAI), vol. 11110, pp. 70–86. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99957-9_5
19. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 229–239. ACM (2015)
20. Johansson, M.: Automated theory exploration for interactive theorem proving. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 1–11. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_1
21. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 291–306. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_21
22. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. J. Autom. Reason. **47**, 251–289 (2010)
23. Johansson, M., Rosén, D., Smallbone, N., Claessen, K.: Hipster: integrating theory exploration in a proof assistant. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) CICM 2014. LNCS (LNAI), vol. 8543, pp. 108–122. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08434-3_9
24. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning. Symbolic Computation (Artificial Intelligence). Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_23
25. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
26. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1–2), 111–156 (2003)
27. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
28. Milder, P., Franchetti, F., Hoe, J.C., Püschel, M.: Computer generation of hardware for linear digital signal processing transforms. ACM Trans. Des. Autom. Electron. Syst. **17**(2), 1–33 (2012)
29. José, M.F., et al.: Spiral: Automatic implementation of signal processing algorithms. In: HPEC, HPEC 2000 (2000)
30. Nandi, C., et al.: Synthesizing structured cad models with equality saturation and inverse transformations. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, pp. 31–44. Association for Computing Machinery, New York (2020)
31. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
32. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. **205**(4), 557–580 (2007)

33. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
34. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI, vol. 50, pp. 1–11. ACM, New York (2015)
35. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. SIGPLAN Not. **50**(10), 107–126 (2015)
36. Ragan-Kelley, J., et al.: Halide: decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**(1), 106–115 (2018)
37. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 80–98. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_5
38. Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. J. Funct. Program. **27**, e18 (2017)
39. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 24–47. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_2
40. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21–25, 2006, pp. 404–415 (2006)
41. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 264–276. Association for Computing Machinery, New York (2009)
42. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: Transit: specifying protocols with concolic snippets. ACM SIGPLAN Not. **48**(6), 287–296 (2013)
43. Valbuena, I.L., Johansson, M.: Conditional lemma discovery and recursion induction in hipster. ECEASST **72**, 1–15 (2015)
44. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
45. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: a language and compiler for DSP algorithms. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 298–308. Association for Computing Machinery, New York (2001)