Quick Theory Exploration for Algebraic Data Types via Program Transformations

Gidon Ernst[1] and Grigory Fedyukovich[2]

[1] LMU Munich, Munich, Germany gidon.ernst@lmu.de
[2] Florida State University, Tallahassee, USA grigory@cs.fsu.edu

Abstract. We present an approach to theory exploration, i.e., a *lemma synthesis procedure* which discovers algebraic laws over recursive functions over Algebraic Data Types (ADTs). The approach, LemmaCalc, builds on, adapts and extends program calculation techniques known from optimization of functional programs (fusion and accumulator removal). Our approach avoids exponential search space of term enumeration (SyGuS) that can render state-of-the-art techniques prohibitively expensive or even useless on large theories with more than a handful of function symbols. In this paper we describe how this approach can be realized and contribute a robust implementation. The evaluation shows that different methods have complementary strengths and that each can produce lemmas not found by the other, but LemmaCalc scales much better to larger theories.

1 Introduction

Algebraic Data Types (ADTs) like applicative lists and trees enable formal modeling of programs in proof assistants like Isabelle/HOL [36] and automatic induction provers, e.g. [52,54,19,29,14,10,38,20,40,31,26]. Equational reasoning and induction are the techniques of choice when proving properties about *recursively-defined functions* over ADTs, usually relying on a set of lemmas that explain what happens if these functions interact with each other. While the libraries of proof assistants usually come with a large amount of lemmas for built-in functions, setting this up for application-specific definitions may incur a significant part of the effort of the verification. A widely-used strategy to achieve a higher-degree of automation are goal-directed proof methods [6], which derive auxiliary lemmas from stuck proofs with the help of generalization heuristics.

Theory exploration is an alternative and complementary direction, aiming to discover lemmas bottom-up from a set of given definitions. State-of-the-art methods [9,42] rely on exhaustive search over millions of candidate formulas in a combinatorial search space by enumeration akin to syntax-guided synthesis (SyGuS) [1]. In SyGuS, usually a vast majority of candidates are either wrong or redundant. Thus, significant efforts in the existing tools, including the state-of-the-art approach TheSy [42], are invested in filtering candidate lemmas using deductive, counterexample-based, or observational-equivalence-based

techniques. Yet, theory exploration in general remains expensive. Furthermore, existing enumeration-based methods generate lemmas that follow no particular pattern or shape and it may be questionable how useful such lemmas are in practice in the context of interactive proofs (i.e., what constitutes progress or simplification of proof goals) and with respect to proof automation (e.g., whether lemmas fit well with proof techniques without introducing matching loops).

Contribution: The key idea behind and advantage of LemmaCalc over search-space enumeration is that it embraces *calculational techniques* based on unfold/fold transformations of recursive functions [7] to guide the search for solutions as well as the underlying induction proof at the same time. In contrast to working at the term/formula level, these approaches transform the definitions of functions themselves, by algorithmically rearranging a given computation into a new form. We emphasize that the effectiveness of LemmaCalc stems from the *combination* of two transformations which to the best of our knowledge is novel: For this work we adapt *fusion* [32] (resp. supercompilation [49], deforestation [50], Sect. 4) and *accumulator removal* by a technique similar to context shift [16] (Sect. 5) to our setting, crucially integrating deduction steps in strategic places. While we have found this combination to be effective our synthesis loop (Sect. 6) may accommodate other transformations too, such as [17,30,27].

Evaluation and Results: We have implemented the approach as an automated tool that takes SMT-LIB files with recursive functions over algebraic data types as input and discovers lemmas that are valid by construction. The evaluation is based on three theories within ADTs: Peano natural numbers, lists, and trees (Sect. 7). To highlight the effect of the combinatorial search space, as an example a naive enumerator would check 320K candidates over 18 list functions and 1M candidates over just 8 functions over natural numbers. When this enumerator as well as state-of-the-art tool TheSy [42] would take *several hours* to cover their large but sparse search spaces, LemmaCalc consistently covers its smaller but targeted search space in a few seconds. Regarding strengths of lemmas found, there is high variability and no single approach is best. Using the method of comparison from [42] (described in Sect. 7), the proportion of lemmas generated by one method and implied by those discovered by another method ranges between $\sim$10% to 100%.

Use and Outlook: We find that LemmaCalc is an effective and efficient method for lemma synthesis that offers a complementary alternative to enumeration-based methods. We envision that the techniques proposed here are particularly suited for larger computer formalizations, e.g. of software systems or mechanized foundations, in which user-defined theories are built on top of already-present libraries. In this scenario, enumeration may be prohibitively expensive, whereas LemmaCalc not only scales better but is well-suited to an incremental workflow.

Data Availability. The implementation of LemmaCalc, the benchmarks, the experimental setup, and instructions to repeat the evaluation are available as an artifact for Linux on Zenodo: `https://doi.org/10.5281/zenodo.16932462`

2 Overview

At a high-level, our approach takes a set of functions $\{f, g, \dots\}$, performs a series of program transformations that rearrange a given computation into new synthetic functions and then relates them among each other and back to the functions originally given. The generated equational lemmas have the form

$$\textbf{fusion:} \qquad f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y}) \tag{1}$$

$$\textbf{accumulator removal:} \qquad f(\overline{x}, a) = e^{?}\big(f'(\overline{x}), \overline{x}', a\big) \tag{2}$$

where f' and fg are synthetic recursive functions (i.e., fg is not just the composition of f and g), and $e^{?}$ is instantiated as an expression. Without loss of generality, to keep the presentation concise, we formalize fusion of g into the last parameter of f and removal of the last accumulator parameter a of f, noting that our tool of course implements the general case.

Lemmas over an original function f can be extracted by recognizing synthetic functions in three possible ways: 1) as the identity function on some argument $x_i \in \overline{x}$, 2) as being equivalent to a recursion-free expression c over a subset $\overline{x}'$ variables, $\overline{x}' \subseteq \overline{x}$, or 3) as being structurally α-equivalent to another function h after permuting its arguments (via some π). That is:

$$\textbf{replacements:} \qquad f(\overline{x}) = x_i \qquad f(\overline{x}) = c(\overline{x}') \qquad f(\overline{x}) = h(\pi(\overline{x})) \tag{3}$$

The role of fusion and accumulator removal is thus to explore different ways to express similar computations, whereas the role of replacements is to discover correspondences that can finally be turned into lemmas.

Running Example: The list ADT is defined over `[]` ("nil"—base constructor) and `::` ("cons"—inductive constructor). Throughout the paper, we illustrate the approach on three recursive functions over lists: `++` ("append"), and `length`, defined in the next two rows, respectively:

$$\texttt{[]} \texttt{++}\ ys \coloneqq ys \qquad (x \texttt{::}\ xs) \texttt{++}\ ys \coloneqq x \texttt{::}\ (xs \texttt{++}\ ys)$$

$$\texttt{length}(\texttt{[]}) \coloneqq 0 \qquad \texttt{length}(x \texttt{::}\ xs) \coloneqq \texttt{length}(xs) + 1$$

In the rest of the section, we illustrate that it is critical to *combine* the respective transformations in LemmaCalc to leverage their full potential. Of the many lemmas discovered for this theory, we now describe how to calculate

$$\texttt{length}(xs \texttt{++}\ ys) = \texttt{length}(xs) + \texttt{length}(ys) \tag{4}$$

The first transformation, **fusion**, merges the recursive traversal in f with that of g by eliminating the intermediate data structure produced by g and consumed by f into a new synthetic function fg.

Example 1. Fusing functions $f = \texttt{length}$ and $g = \texttt{++}$ e.g. by the approach of [18] or [37], calculates the following definition for synthetic function $fg = \texttt{length}_{\texttt{++}}$.

Starting from the left-hand side of the desired equation, $\texttt{length}(xs \mathbin{+\!\!+} ys)$, we discern the two cases of the definition of g,

$$\begin{aligned}\texttt{length}_{+\!\!+}([\,], \underline{ys}) &= \texttt{length}([\,] \mathbin{+\!\!+} ys)\\ &= \underline{\texttt{length}(ys)}\end{aligned}$$

$$\begin{aligned}\texttt{length}_{+\!\!+}(x :: xs, \underline{ys}) &= \texttt{length}((x :: xs) \mathbin{+\!\!+} ys) = \texttt{length}(x :: (xs \mathbin{+\!\!+} ys))\\ &= \texttt{length}(xs \mathbin{+\!\!+} ys) + 1 = \texttt{length}_{+\!\!+}(xs, \underline{ys}) + 1\end{aligned}$$

for which $\texttt{length}(xs \mathbin{+\!\!+} ys) = \texttt{length}_{+\!\!+}(xs, ys)$ by construction. Note, the difference in definitions of $\texttt{length}_{+\!\!+}$ and $\texttt{length}$ is underlined: the entire base case and additional parameter ys, which is passed unchanged to the recursive call. ■

Fusion tends to regularize the way in which computations are laid out. Complementary, **accumulator removal**, "straightens" the computations but in a different form, by relating functions with accumulators and those without them (we regard ys in Ex. 1 as an accumulator, too). Recall (2): removal of accumulator a in $f(\overline{x}, a)$ gives a synthetic function f' that mirrors f and an expression $e^?$ over the outputs of f', a, and $\overline{x}' \subseteq \overline{x}$. Removing accumulators is useful both for original and synthetic functions. Specifically, fused functions $fg(\_, \overline{y})$ tend to retain some of the arguments $\overline{y}$ of g as accumulators such as ys in Ex. 1:

Example 2. Removing accumulator ys from $\texttt{length}_{+\!\!+}$ yields f' as $\texttt{length}'_{+\!\!+}$

$$\texttt{length}'_{+\!\!+}([\,]) \coloneqq 0 \tag{5}$$

$$\texttt{length}'_{+\!\!+}(x :: xs) \coloneqq \texttt{length}'_{+\!\!+}(xs) + 1$$

$$\text{so that}\quad \texttt{length}_{+\!\!+}(xs, ys) = \texttt{length}'_{+\!\!+}(xs) + \underline{\texttt{length}(ys)} \tag{6}$$

where the underlined part is the base case expression from Ex. 1 and $e^?$ in (2) is instantiated by $+$. This solution can be found algorithmically as described in Sect. 5 by relying on neutral elements like 0 to replace base case expressions like in (5) and use the respective operator like $+$ in the solution for $e^?$. As it turns out, $\texttt{length}'_{+\!\!+}$ is structurally equivalent to $\texttt{length}$ and we can apply a corresponding **replacement** schema—here the third case in (3)—to Eq. (6) to conclude Eq. (4) as a lemma found by the approach. ■

3 Preliminaries

In this work, we rely on a first-order, many-sorted functional specification language, which includes inductive algebraic data types (ADTs) like lists and trees. A typed n-ary function $f\colon t_1, \ldots, t_n \to t$ is presented as a

$$\textbf{function definition}\qquad f(\overline{p}_1) \coloneqq e_1 \texttt{ if } \varphi_1 \quad \cdots \quad f(\overline{p}_m) \coloneqq e_m \texttt{ if } \varphi_m$$

comprised of a set of m cases, each with pattern $\overline{p}_i = p_1, \ldots, p_n$, a boolean expression as guard φ_i and the right-hand side e_i, for each $0 < i \leq m$. A case

is *recursive* if the right-hand side e_i or the guard φ_i contains calls to f. We call functions f and g supplied by the user *original* whereas intermediate definitions that are generated algorithmically are called *synthetic*, typically denoted with a prime (e.g. f') or pairs of names (e.g. $\texttt{length}_{+\!+}$ used in the previous section).

Notation and Conventions. We require that all case-distinctions are expressed at the top-level using guards, i.e., explicit if-then-else and case-of expressions have been transformed away (this is always possible). This means that the grammar for expressions e and e' just consists of variables x, and the applications of function symbols f, f', and g and *constructor* symbols c and d, patterns p and q contain no defined functions, and values v are (possibly nested) constructor terms, where $\overline{v}$ can again have constructors but no variables. That is

$$\begin{array}{rlll} \textbf{expressions} & e, e' \coloneqq x \mid c(\overline{e}) \mid f(\overline{e}) & & \text{as well as} \\ \textbf{patterns} & p, q \coloneqq x \mid c(\overline{p}) & \text{and} & \textbf{values} \quad v \coloneqq c(\overline{v}) \end{array}$$

By $\mathit{free}(e)$ we denote the set of free variables of e. A substitution σ is a mapping from variables to expressions, writing $\sigma(e)$ for applying it to e. Oriented left-to-right, equations from the definitions as well as the lemmas discovered can be interpreted as conditional *rewrite rules*. Let Γ be a set of definitional equations and lemmas, we write $\Gamma \vdash e \rightsquigarrow e'$ if expression e can be rewritten to e' by applying a finite number of definitions and lemmas in Γ while showing that the respective side-conditions follow from Γ. Rewriting is assumed to be soundly implemented, i.e, all models of Γ validate $e = e'$.

Assumptions & Scope. We rely on the following assumptions on the original functions, which are typical for definitions in inductive theorem provers,[3] and all synthetic functions generated by our constructions will retain these properties. All functions are terminating under strict evaluation, i.e., each function f is equipped with a corresponding well-founded order $\prec_f$ that connects arguments to recursive calls, i.e., for a recursive case $f(\overline{p}) \coloneqq e\big(\, f(\overline{e}) \,\big)$ `if` φ of the definition of f satisfies $\forall \overline{x}.\ \varphi \implies \overline{e} \prec_f \overline{p}$ where $\overline{x} = \mathit{free}(\overline{p})$ are the variables in scope. Constructions in this paper are justified by induction on these orders. We require that the patterns together with guards disjointly partition the entire set of possible arguments, i.e., functions are total, and the order of matching cases is irrelevant. That is, we can represent the definition as a consistent set of logical axioms and define transformations case-by-case.

The approach presented in this paper as well as our implementation assumes that there are no nested recursive calls and there are no mutually recursive definitions. We assume that there are no recursive calls in guards and that the bodies of definitions are quantifier-free. Finally, our approach is *essentially* first-order, but we support the SMT-LIB style functional arrays as parameters to "higher-order" functions like `map`, `filter`. Lifting these limitations future is work.

[3] Isabelle/HOL ensures these properties even if some aspects are transparent to the user, by inferring termination orders, by translating sequential pattern matches into disjoint, parallel ones, and by replacing underspecification by a constant `undefined`.

Theory Exploration. The underlying idea is to generate lemmas bottom-up from a set of definitions instead of taking these from intermediate proof goals.

Definition 1 (Theory Exploration). *Given a set F of typed functions and predicates $f : t_1, \ldots, t_n \to t \ \in \ F$, and given a set Δ of axioms which define the functions, compute a set Λ of lemmas, so that $\Delta \models \Lambda$, i.e., each model for the F that satisfies all axioms Δ is also a model of each lemma in Λ.*

We emphasize that this definition on its own only guarantees correctness of the lemmas found, but not their utility, which may be tricky to characterize. In practice, theory exploration methods apply heuristics to filter out trivial and redundant lemmas, an aspect that becomes relevant in the evaluation.

Given function definitions, known lemmas Γ about them, and a proof oracle that semi-decides $\Gamma \vdash \varphi$, theory exploration can thus be formulated to find a valid $\varphi^?$ from a search space $\Sigma(\overline{x}, \texttt{bool})$ of candidate formulas:

$$\textbf{theory exploration} \qquad \Gamma \vdash \varphi^? \qquad \text{where } \varphi^? \in \Sigma(\overline{x}, \texttt{bool}) \tag{7}$$

We can impose a certain form for $\varphi^?$ to restrict the search space. For instance, our baseline enumerator considers candidates for $f(\overline{x}, g(\overline{y})) = rhs^?$ specifically to match the shape of lemmas generated by our approach (Sect. 2) to measure its effectiveness in relation to the search space.

Baseline: Enumerative Synthesis. Given a set F of typed functions/predicates $f : t_1, \ldots, t_n \to t \ \in \ F$ we can define the search space $\Sigma_d(\overline{x}, t)$ of terms of type t over typed variables $\overline{x}$ up to depth d recursively. The terms of depth 0 are just the variables of matching type, whereas in the recursive case, for each function f from F we enumerate possible arguments of smaller depth.

$$\begin{aligned} \Sigma_0(\overline{x}, t) &= \{x_i \in \overline{x} \mid x_i : t\} \\ \Sigma_d(\overline{x}, t) &= \{\ f(e_1, \ldots, e_n) \mid f : t_1, \ldots, t_n \to t \in F \text{ and} \\ &\qquad\qquad e_i \in \Sigma_{d_i}(\overline{x}, t_i) \text{ for } d_i < d, i = 1, \ldots, n\ \} \end{aligned} \tag{8}$$

Empirically, it is a good strategy to limit the number of occurrences o of each variable. For the theories considered in this paper, choosing $o = 2$ or $o = 3$ cuts down the search space significantly while still retaining all "reasonable" lemmas (i.e., those that one would use in practice, see also Sects. C and 7).

$$\Sigma^o_d(\overline{x}, t) = \{\ e \in \Sigma_d(\overline{x}, t) \mid \text{each } x_i \in \overline{x} \text{ occurs max. } o\text{-times in } e\ \}$$

4 Fusion

Fusion of two functions f and g aims to compute a synthetic function fg such that $f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y}))$ is valid by construction. We first introduce the notion of a *fused form* that guarantees that each recursive call of f over a recursive one of g has been merged into a joint recursive call to fg. The intuition is that fused form captures when elimination of the intermediate result of g is possible.

Algorithm 1: Algorithm $\textsc{Fuse}(\Gamma, f, g)$. Without loss of generality, g is fused into the last argument of f.

Input: Γ, set of definitions and known lemmas, including
$\{\ f(\overline{p}_i^f, q_i^f) \coloneqq e_i^f \ \texttt{if}\ \varphi_i^f\ \}_{i=1,\ldots,m} \subseteq \Gamma$ and
$\{\ g(\overline{p}_j^g) \coloneqq e_j^g \ \texttt{if}\ \varphi_j^g\ \}_{j=1,\ldots,n} \subseteq \Gamma$

Output: Φ with def. of fg and lemma $f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y})$

1 $\Phi \leftarrow \{\ f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y})\ \}$
2 **for** $j \leftarrow 1, \ldots, n$ cases in the definition of g **do** — unfold g
3 $\quad \Gamma_{fg} \leftarrow \Gamma \cup \{\ \overline{y} \prec_g p_j^g \implies f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y})\ \}$
4 $\quad$ **if** $\Gamma \vdash f(\overline{x}, e_j^g) \rightsquigarrow e'$ *and* e' *is in fused form wrt.* f *and* g **then** — fold fg?
5 $\quad\quad \Phi \leftarrow \Phi \cup \{\ fg(\overline{x}, \overline{p}_j^g) \coloneqq e' \ \texttt{if}\ \varphi_j^g\ \}$
6 $\quad$ **else**
7 $\quad\quad$ **for** $i \leftarrow 1, \ldots, m$ cases of f **do** — unfold f
8 $\quad\quad\quad$ **assert** $\mathit{free}(\overline{p}_i^f) \cap \mathit{free}(\overline{p}_j^g) = \varnothing$ (possibly rename)
9 $\quad\quad\quad$ **if** $\Gamma \vdash e_j^g \rightsquigarrow e'$ *and* $\exists\ \sigma.\ q_i^f \overset{\sigma}{\equiv} e'$ *and* $\Gamma_{fg} \vdash \sigma(e_i^f) \rightsquigarrow e''$ *so that* e'' *is in fused form wrt.* f *and* g **then** — fold fg?
10 $\quad\quad\quad\quad$ **let** $\overline{p} \leftarrow \sigma(\overline{p}_i^f, \overline{p}_j^g)$ and $\varphi \leftarrow \sigma(\varphi_i^f \wedge \varphi_j^g)$
11 $\quad\quad\quad\quad \Phi \leftarrow \Phi \cup \{\ fg(\overline{p}) \coloneqq e''\ \texttt{if}\ \varphi\ \}$
12 $\quad\quad\quad$ **else if** $q_i^f \perp e'$ **then**
13 $\quad\quad\quad\quad$ (case i of f cannot match result of case j of g)
14 $\quad\quad\quad\quad$ **continue**
15 $\quad\quad\quad$ **else**
16 $\quad\quad\quad\quad$ (indeterminate or blocked due to missing lemma)
17 $\quad\quad\quad\quad \Phi \leftarrow \varnothing$ and **fail**

Definition 2 (Fused form). *An expression e is in* fused form *with respect to f and g if g does not occur nested in any argument of f anywhere in e.*

Definition 3 (Pattern Unification and Refutation). *Assuming that free variables of p are disjoint to those of e, we write $p \overset{\sigma}{\equiv} e$ when substitution σ is the most general unifier of pattern p and expression e [41, Def. 5.9]. We write $p \perp e$ when there can be no such unifier, i.e., the pattern match is "refuted".*

$$p \overset{\sigma}{\equiv} e \iff \big(\exists\ \sigma.\ \sigma(p) = \sigma(e)\big) \wedge \big(\forall\ \sigma'.\ \sigma'(p) = \sigma'(e) \Rightarrow \exists\ \sigma''.\ \sigma' = \sigma'' \circ \sigma\big)$$
$$p \perp e \iff \big(\forall\ \sigma.\ \sigma(p) \neq \sigma(e)\big)$$

The fusion algorithm, Alg. 1, is realized as an unfold/fold transformation [7]. Conceptually, it lets $fg(\overline{x}, \overline{y}) \coloneqq f(\overline{x}, g(\overline{y}))$ and then transforms the right-hand side into the fused form. Algorithmically, it pairs each defining j-th case of g with each i-th case of f, by analyzing how the result returned by g via body expression e_j^g can be matched by pattern q^f of the fused argument position. The case analyses correspond to "unfolds", cf. line 3 for g and line 7 for f (we discuss the optimization in line 4 shortly). Line 9 checks whether the pairing is feasible

by computing the most general unifier (cf. Def. 3) between g's result and f's pattern, and if so, adds a corresponding defining case for fg.

The main concern is that the expression e'', which is ultimately used in the definition of fg (line 11), is in the fused form wrt. f and g. To achieve this we can make use of folding that collapses joined recursive calls $f(\_, g(\_))$ into recursive fg-calls (applied in lines 4 or 11). Technically, it is realized by a *fold rule*, added to the set of known facts Γ in line 3 (we discuss its premise below).

Our presentation makes explicit the way in which fusion is intertwined with the application of definitions and facts already known by rewriting wrt. Γ. A key optimization is in line 4, which avoids unfolding f altogether when the case of g is non-recursive as in Ex. 1 where the base case wraps $e^g = ys$ directly by $f = \texttt{length}$. It applies to tail-recursive cases of g, too, which are immediately folded in line 4 by the additional rule in Γ_{fg}, and when lemmas help to eliminate g altogether. In practice, this optimization crucially retains the structure needed to recognize fused functions and their derivatives in terms of original ones.

Premise $\overline{y} \prec_g p_j^g$ of the fold rule in line 3 ensures that recursive fg calls respect the termination order $\prec_g$ of g despite the rewriting steps in lines 4 and 9 (cf. Sect. B.1). In the evaluation, this premise is always satisfied.

Example 3. Alg. 1 works analogously to the calculations in Ex. 1, but instead of induction on a single argument it works alongside the cases of the inner function g (here the two notions coincide). Unfolding $+\!\!+$, we have one base case ($j = 1$) and one recursive case ($j = 2$), and to illustrate wrt. the above calculation we have $\overline{p}_1^g = \texttt{[]}, ys$ and $\overline{p}_2^g = x :: xs, ys$.

In the base case, moreover e_1^g is ys concretely so that $f(e_1^g)$ is $\texttt{length}(ys)$, which is already in fused form (line 4), leading to the base case of $\texttt{length}_{+\!\!+}$ as shown above and in Ex. 1. Note, if we were to unfold $\texttt{length}$ as well in this situation, we would instead get *two* cases, as e' being ys unifies with both patterns $\texttt{[]}$ and $x :: xs$ of $\texttt{length}$ in line 9. Apart from destroying the correspondence between $\texttt{length}_{+\!\!+}$ and $\texttt{length}$ it turns that argument into a non-accumulator and thus prevents progress later on.

In the recursive case $e_2^g = x :: (xs +\!\!+ ys)$ and $\texttt{length}(e_2^g) = \texttt{length}(xs +\!\!+ ys) + 1$ by definition. We make use of the fold rule (line 4), which here is

$$xs \prec_{+\!\!+} (x :: xs) \implies \texttt{length}(xs +\!\!+ ys) = \texttt{length}_{+\!\!+}(xs, ys)$$

in which the premise holds so that $e'' = \texttt{length}_{+\!\!+}(xs, ys)+1$ in fused form becomes the right-hand side of the recursive case. ■

Ex. 7 in Sect. A further details how Alg. 1 relies on known lemmas from Γ to make progress in line 9 to achieve fused form.

5 Accumulator Removal

Accumulator removal aims to express $f(\overline{x}, u)$ as $e^?(f'(\overline{x}), \overline{x}', u)$ for a synthetic function f' that in comparison to f lacks accumulator u (Def. 4). Expression $e^?$

compensates for the absence of u in the computation of f', such that the definition of f' and $e^?$ must be found hand-in-hand. This $e^?$ may depend on the accumulator and the "static" subset $\overline{x}'$ of the remaining arguments (Def. 5).

To make the recursive calls in the body of a function explicit, we denote each i-th case $f(\overline{p}_i, u) \coloneqq e_i$ `if` φ_i of f as a decomposition into a "body" expression b_i that makes k recursive calls with regular arguments e_i^j and computes the new value for the accumulator using expressions $a_i^j(u)$.

$$e_i = b_i\big(\, f\big(e_i^1, a_i^1(u)\big), \ldots, f\big(e_i^k, a_i^k(u)\big)\,\big) \tag{9}$$

Definition 4 (Accumulator). *A parameter is an accumulator of f if 1) it is matched by a variable u in each pattern, 2) it specifies the values a_i^j for the same argument position of recursive calls in recursive cases, 3) it does not occur in guards φ_i or elsewhere in any recursive body b_i (u may be used in base cases).*

Definition 5 (Static Parameter and Expressions). *A parameter is called* static *if it is passed by identity only, $a_i^j(u) = u$, A subexpression of a function definition is static if it depends on static parameters only.*

Static subexpressions retain their value when lifted out of the recursion to the top-level. As an example, *ys* of $\texttt{length}_{+\!+}$ in Sect. 2 is static.

Alg. 2 shows our algorithm for accumulator removal. It generates f' case-by-case from the definition of f by matching its recursive structure, but by allowing different body expressions b_i'. Note patterns p_i and guards φ are kept to preserve the recursive traversal, so that arguments match the i-th case of f exactly iff they match the i-th case of f'. For that reason, the removed accumulator may not occur in guards in the first place (cf. Def. 4). The algorithm is effectively a straight-forward translation of an inductive proof of the desired lemma. Note that it is conceptually analogous to Giesl's context transformations [16], but it is formulated in a more straight-forward way.

The algorithm is presented nondeterministically here. The key first choice occurs in line 7, where a solution for critical base cases is chosen, i.e., those base cases which refer to the accumulator u, cf. $\texttt{length}_{+\!+}(\texttt{[]}, ys) = \texttt{length}(ys)$ for *ys* from Ex. 1. The heuristic we adopt is to pick b_i in $f'(p) = b_i$ to be the neutral element c of a binary function/operator $\oplus$. We shift the entire original body b_i out of the function as part of $e^?$, noting that references to static parameters *xs* within b_i retain their meaning over the shift. For Ex. 2, the correct choice is $\oplus$ as $+$ with neutral element $c = 0$ and therefore $e^?(y) = y + \texttt{length}(u)$, however, our implementation tries other combinations like $\times$ and 1 and backtracks when line 14 is hit (this is the only place where we use trial and error).

The second key choice is in line 10, where we pick a body b_i' for other all other (base and recursive) cases from the search space $\Sigma_d(\overline{z}, t_r)$ of expressions of f's return type over the variable in scope $\overline{z}$ (recall its inductive definition in (8)). The condition checked in line 13 ensures that the choice is compatible with any (prior) choice of $e^?$, i.e., that $e^?$ commutes from inside recursion all the way to the top-level of the lemma to be synthesized.

Algorithm 2: Algorithm RemoveAcc(Γ, f), presented without loss of generality with the accumulator as the last argument of f.

Input: Γ, set of definitions and known lemmas
Input: definition of $f\colon \overline{t}, t_u \to t_r$ as $\{\ f(\overline{p}_i, u) \coloneqq e_i \texttt{ if } \varphi_i\ \}_{i=1,\dots,m} \subseteq \Gamma$
Output: Φ with of the definition of f' and $f(\overline{x}, u) = e^?(f'(\overline{x}), \overline{x}', u)$

1 $\Phi \leftarrow \{\ f(\overline{x}, u) = e^?(f'(\overline{x}), \overline{x}', u)\ \}$
2 **for** $i \leftarrow 1, \dots, n$ (cases in the definition of f) **do**
3 **let** $\overline{z} = \mathit{free}(\overline{p}_i)$ be the free variables in $\overline{p}_i$
4 **let** $\overline{x}'\colon \overline{t}'$ be the static arguments in $\overline{p}_i$
5 **let** $b_i\big(\ f\big(e_i^1, a_i^1(u)\big), \dots, f\big(e_i^k, a_i^k(u)\big)\ \big) = e_i$ *(cf. (9))*
6 **if** $k = 0$ (base case) and $u \in \mathit{free}(b_i)$ and $\mathit{free}(b_i) \setminus u \subseteq \overline{x}'$ **then**
7 **choose** binary $\oplus$ with neutral element c from $\Gamma \vdash \forall z.\ c \oplus z = z$
8 $b_i' \leftarrow c$ and $e^?(y, \overline{x}', u) \leftarrow y \oplus b_i$
9 **else**
10 **choose** $b_i' \in \Sigma_d(\overline{z}, t_r)$ (such as b_i if $u \notin \mathit{free}(b_i)$)
11 $\mathit{lhs} \leftarrow b_i\big(\ e^?\big(y^1, \overline{x}', a_i^1(u)\big), \dots, e^?\big(y^k, \overline{x}', a_i^k(u)\big)\ \big)$
12 $\mathit{rhs} \leftarrow e^?\big(b_i'\big(\ y^1, \dots, y^k\ \big), \overline{x}', u\big)$
13 **if** *not* $\Gamma \vdash \forall\ \overline{y}, \overline{z}, u.\ \varphi_i \implies \mathit{lhs} = \mathit{rhs}$ **then**
14 **fail**
15 $\Phi \leftarrow \Phi \cup \{\ f'(p_i) \coloneqq b_i'\big(\ f(e_i^1), \dots, f(e_i^k)\ \big) \texttt{ if } \varphi_i\ \}$

Example 4. To continue Ex. 2 for the accumulator removal of $f = \texttt{length}_{+\!+}$ works with the sketch for $f' = \texttt{length}'_{+\!+}$

$$\begin{aligned} \texttt{length}'_{+\!+}(\texttt{[]}) &\coloneqq b_1 \\ \texttt{length}'_{+\!+}(x :: xs) &\coloneqq b_2(x, xs, \texttt{length}'_{+\!+}(xs)) \\ \text{so that}\quad \texttt{length}_{+\!+}(xs, ys) &= e^?(\texttt{length}'_{+\!+}(xs), ys) \end{aligned}$$

and finds instances for b_1, b_2, and $e^?$ as follows.

The base-case condition $\texttt{length}(ys) = e^?(b_1', ys)$ is solved by choosing c and $\oplus$ as left-neutral element 0 of $+$, i.e., we have $\forall z.\ 0 + z = z$. For $b_i = \texttt{length}(ys)$, we therefore instantiate $b_1' = 0$ and $e^?(zs, ys) = zs + \texttt{length}(ys)$. For the recursive case, we (heuristically) preserve $b_2' = b_2 = \_ + 1$. The condition checked in line 13 is valid: $\forall m, ys.(m + \texttt{length}(ys)) + 1 = (m + 1) + \texttt{length}(ys))$. ■

Example 5. Associativity of $+\!+$ is discovered by fusing $(xs +\!+ ys) +\!+ zs$ into a synthetic function $+\!+\!+(xs, ys, zs)$ and then by removing its last argument. The correct choices are $b_1' = c = \texttt{[]}$ as *right*-neutral element of $\oplus = +\!+$ and canonically $b_2' = b_2 = \_ :: \_$ as the original function body of the outer $\_ +\!+ zs$. ■

Example 6 (Reversing Lists). We show how accumulator removal is key to the classic lemma $\texttt{reverse}(\texttt{reverse}(xs)) = xs$ in our approach

$$\texttt{reverse}(\texttt{[]}) = \texttt{[]} \qquad \texttt{reverse}(x :: xs) = \texttt{reverse}(xs) +\!+ (x :: \texttt{[]})$$

Because `reverse` has no accumulators, we start with fusion, which fails initially for `reverse(reverse(_))` as we cannot match $+\!\!+$ in the body of the inner $g =$ `reverse` (there is no unifier nor can we refute the match). However, from fusing $+\!\!+$ into `reverse` we get:

$$\begin{aligned}\texttt{reverse}_{+\!\!+}(\texttt{[]}, ys) &= \texttt{reverse}(ys)\\ \texttt{reverse}_{+\!\!+}(x :: xs, ys) &= \texttt{reverse}_{+\!\!+}(xs, ys) +\!\!+ (x :: \texttt{[]})\end{aligned}$$

This function has an accumulator that can be removed, again with $\oplus = +\!\!+$ but now with $c =$ `[]` as its *left*-neutral element, resulting in $\texttt{reverse}(xs +\!\!+ ys) = \texttt{reverse}(ys) +\!\!+ f'(xs)$ and it turns out that $f' \equiv$ `reverse`. This lemma unblocks fusion of `reverse(reverse(_))` via the shortcut in line 4 of Alg. 1. The fused function `reverse_reverse` can subsequently be recognized as the identity function after some simplifications. ■

Another classic example is the tail-recursive function `qreverse`, shown in Ex. 8 in Sect. A. It requires a more complex choice for b_i' in line 10 of Alg. 2 and is not discovered by our implementation, but is in generally in reach of our approach.

6 Main Algorithm

Alg. 3 saturates a database Γ of definitions and discovered lemmas by repeatedly applying the transformation on original as well as synthetic functions. Algorithms Fuse (cf. Sect. 4) and RemoveAcc (cf. Sect. 5) return an equation of the corresponding shape as shown above if they succeed, together with the defining equations of the respective synthetic functions.

Algorithm 3: Lemma synthesis by saturation using fusion, accumulator removal, and recognition of structurally similar functions.

Input: Set Δ of definitions of the original functions
Output: Set Λ of lemmas discovered over Δ

1 $\Gamma \leftarrow \Delta$
2 **repeat**
3 $\quad\Gamma \leftarrow \Gamma \cup \textsc{Fuse}(\Gamma, f, g)$ for pairs of functions f, g
4 $\quad\Gamma \leftarrow \Gamma \cup \textsc{RemoveAcc}(\Gamma, f)$ for f with accumulator
5 $\quad\Gamma \leftarrow \Gamma[f(\overline{x}) \mapsto rhs]$ for replacements $f(\overline{x}) = rhs$
6 $\Lambda \leftarrow \textsc{Extract}(\Gamma) \setminus \Delta$

The three steps (fusion, accumulator removal, conditional lemmas) may benefit from the accumulated set of lemmas Γ, as shown in Ex. 7 for fusion. Our algorithm retries failed steps as long as new information can be gained. Intermittently, the algorithm applies replacement lemmas (3) eagerly. This de-duplicates

the effort and avoids vacuously fused forms (cf. Sect. 4). Replacement is oriented to keep original functions if possible. The final step of the algorithm is to extract useful lemmas from Γ:

$$\textsc{Extract}(\Gamma) = \{\ \varphi'' \mid \text{for lemma } \varphi \in \Gamma \text{ where } \Gamma \vdash \varphi \rightsquigarrow \varphi' \text{ and } \mathit{recover}(\Gamma) \vdash \varphi' \rightsquigarrow \varphi'' \text{ and } \varphi'' \text{ uses original functions only } \}$$

where $\mathit{recover}(\Gamma) = \{\ fg(\overline{x},\overline{y}) = f(\overline{x}, g(\overline{y})) \mid f(\overline{x}, g(\overline{y}) = fg(\overline{x},\overline{y}) \in \Gamma\ \}$ recovers fused functions in terms of their original sources; done in a separate step to avoid rewrite loops between the symmetric rules in Γ and $\mathit{recover}(\Gamma)$.

Lemma 1 (Soundness of Transformations). *Both transformations* Fuse *and* RemoveAcc *produce valid lemmas, and new synthetic functions satisfy all assumptions of Sect. 3 (the proofs are in Sect. B).* □

Theorem 1 (Soundness of Alg. 3). *All lemmas computed are valid wrt. the original definitions* $\Delta \models \Lambda$. □

Final remark: As we are relying on rewriting as our main technique to apply definitions and lemmas, we briefly address the question of potentially looping rewrite rules. A general technique to detect and avoid nontermination is [11]. Alternatively, one can represent Γ as an E-graph [12,35,51], which can accommodate cyclic expressions and is therefore more robust against this issue. In our experiments, however, the only cases for rewrite loops come from lemmas produced by accumulator removal at some intermediate stages and it was sufficient to not use these lemmas as long as they still contain synthetic functions.

7 Evaluation

We implemented LemmaCalc in a fully automated tool and evaluated it on three theories within ADTs. We used the Z3 SMT solver (v4.12.4) [34] as the proof oracle to decide entailments $\Gamma \vdash \varphi$. The goal of this evaluation is to substantiate that LemmaCalc is a practical and effective procedure for theory exploration and to understand its strengths and limitations:

- **RQ1**: What is the relative explanatory strength of the sets of lemmas generated by the different methods?
- **RQ2**: What is the impact of the search space on lemma synthesis time?

Experimental Setup. We compare LemmaCalc with two alternatives:

- **Enum**: Our own enumerative generator, based on (7) in Sect. 3, which solves for $e^?$ in equational lemmas $f(\_, g(\_)) = e^?$ without preconditions
- **TheSy**, a state-of-the-art enumerative lemma generator [42] using E-graphs. TheSy in contrast to our baseline can discover conditional lemmas.

We had originally intended to compare against HipSpec [9], too, but that was not possible due to technical issues with its installation.

The baseline enumerator (Enum) is included because it provides (approximate) ground truth on the lemmas that can possibly discovered by LemmaCalc using fusion and removal of accumulators. In our experience on the benchmarks, deeply nested lemmas are usually redundant. Thus, Enum explores a search space up to depth $d = 3$ and maximal variable occurrence $o = 2$ in (7), which is sufficient to cover all lemmas found by LemmaCalc. Solver timeout was configured to 1000ms per query for the baseline evaluator.

As the search space is huge, before proof attempts, Enum relies on a ground evaluator to exhaustively search for counterexamples to lemma candidates of a small size. It makes a large effect when false formulas are filtered out—much faster and more reliable than using Z3. To discover any non-trivial lemma, we enrich the proof oracle by an induction preprocessing step that in turn tries all potential induction variables. For example, $\forall n\colon \texttt{nat}.\ P(n)$ is passed as $P(0) \wedge (\forall n\colon \texttt{nat}.\ P(n) \implies P(n+1))$ to the proof oracle.

All methods runs multiple rounds of lemma discovery so that proofs that had failed earlier can benefit from lemmas discovered later (Examples 6 and 7).

The evaluation is based on three theories, over Peano arithmetic, and over functional lists and trees, respectively. In addition to the full theories (`nat`, `list`, and `tree` below), we consider eight benchmarks with a subset of functions that together make up some interesting lemmas. As shown in Table 1, a full theory gets from 5 to 18 functions, from which our baseline enumerator generates ∼1.5M candidates in total, of which roughly ∼0.01% are true lemmas only (we comment on the run times below). Some details on the benchmarks are in Sect. C. Experiments were run on a Lenovo T470 Thinkpad with 4x Intel(R) Core(TM) i5-7440HQ CPU @ 2.80GHz and 32 GB main memory.

Method of Comparison. A benchmark consists of a set of definitions Δ, from which a lemma synthesis method generates a set Λ of lemmas so that $\Delta \models \Lambda$. For **RQ1** we are interested in a comparison in terms of relative explanatory strength of these sets Λ as discussed in depth in [42].

Definition 6 (Subsumption). *For a set of lemmas Λ_A generated by one method, the subset of Λ_A that is "subsumed" by Λ_B is $\mathcal{S}(\Lambda_A, \Lambda_B) = \{\varphi \in \Lambda_A \mid \Delta, \Lambda_B \vdash \varphi\}$ where $\Gamma \vdash \varphi$ denotes that φ is provable by an oracle given facts Γ.*

The ratio $|\mathcal{S}(\Lambda_A, \Lambda_B)|/|\Lambda_A|$ can therefore be understood as a proxy for the proportion of the knowledge that can be gained from Λ_A that can also be gained from Λ_B [42]. In practice, however, the generated sets of lemmas tend to contain some trivial lemmas (e.g. that follow from Δ without induction) and some redundancies (e.g. lemmas that are implied from the others). These aspects are not adequately captured by subsumption alone. To give an example of an effect that we have observed, if method A discovers commutativity of $+$ on numbers but method B does not, then Λ_A from Def. 6 contains two equivalent lemmas $\varphi(a+b)$ and $\varphi(b+a)$ for each suitable φ, a, and b. Moreover, many of the benchmarks contain functional and predicate symbols from the background theory,

e.g., $+$, $<$, $\leq$, $\neg$, but we do not want to count lemmas over just these as part of more complex benchmarks. Therefore, for each set of lemmas Λ generated with respect to a given Δ, where F_0 are the background functions, we define

- $\mathcal{L}(\Lambda) = \{\varphi \in \Lambda \mid F_0 \subset \mathit{funs}(\varphi)\}$ is the relevant set of lemmas generated by the tool, the remaining ones $\mathcal{B}(\Lambda) = \Lambda \setminus \mathcal{L}(\Lambda)$ are the "background" lemmas.
- $\mathcal{N}(\Lambda) = \{\varphi \in \mathcal{L}(\Lambda) \mid \Delta, \mathcal{B}(\Lambda) \nvdash \varphi\}$ is the "non-trivial" subset of the non-background lemmas. Note, we choose to exclude also those consequences that are made true by the background lemmas (such as consequences of commutativity of $+$ as discussed above).
- $\mathcal{R}(\Lambda) \subseteq \mathcal{L}(\Lambda)$ with $\Delta, \mathcal{R}(\Lambda) \vdash \mathcal{L}(\Lambda) \setminus \mathcal{R}(\Lambda)$ is a "reduced" set of lemmas after removing some redundant ones. [4]

For any benchmark, for each lemma synthesis method A, we report the cardinalities of $\mathcal{L}(\Lambda_A) \supseteq \mathcal{N}(\Lambda_A) \supseteq \mathcal{R}(\Lambda_A)$, as well as the respective subsumptions for all other methods B, namely $\mathcal{S}(\mathcal{L}(\Lambda_A), \Lambda_B) \supseteq \mathcal{S}(\mathcal{N}(\Lambda_A), \Lambda_B) \supseteq \mathcal{S}(\mathcal{R}(\Lambda_A), \Lambda_B)$. Solver timeout was configured to 100ms per query in the comparison to keep evaluation times tractable.

RQ1: What is the relative explanatory strength of the sets of lemmas generated? This comparison is shown in Fig. 1—note that the range for the y-axis varies across the benchmarks to aid readability.

As an example, on benchmark `append`, the structural method finds 9 lemmas in $\mathcal{L}$, of which 4 are redundant (i.e. just in $\mathcal{N}$ and filtered out by $\mathcal{R}$) whereas the other 5 are in the reduced set, and there are no trivial lemmas. Enumeration and TheSy cover 3 of the 4 redundant and 4 resp of the 5 reduced lemmas each. Enum misses distributivity of *count* over `++`, this lemma is part of the final "unknown" lemmas despite having a straight-forward inductive proof. We conjecture that the solver enters a matching loop due to presence of commutativity of addition and therefore times out. On the other hand, the three lemmas uniquely found by Enum are enabled due to this commutativity in the first place. Such effects are likely to be present in other benchmarks, too. TheSy moreover misses associativity of `++` for unknown reasons.

Overall, the performance of all methods is usually in the same order of magnitude, but the specific results differ widely across the benchmarks. In absolute terms, methods based on enumeration may be seen to to outperform LemmaCalc, as represented by the relatively larger first bar in the respective column, noticeable e.g., on benchmark `length`. This is expected as they cover a much larger space. However, calculational techniques can cover a significant proportion of that space, and also generate lemmas not found by the other approaches.

On benchmarks `filter` and `remove`, most resp. all interesting lemmas are conditional equations, recall that these cannot be generated by LemmaCalc and Enum. On `filter`, the lemmas not found by our methods are for example related to filtering twice with the same predicate (e.g., duplicate occurrence of a variable, a limitation of Alg. 1).

[4] Note this set is not unique. In the evaluation we used a greedy incremental algorithm.

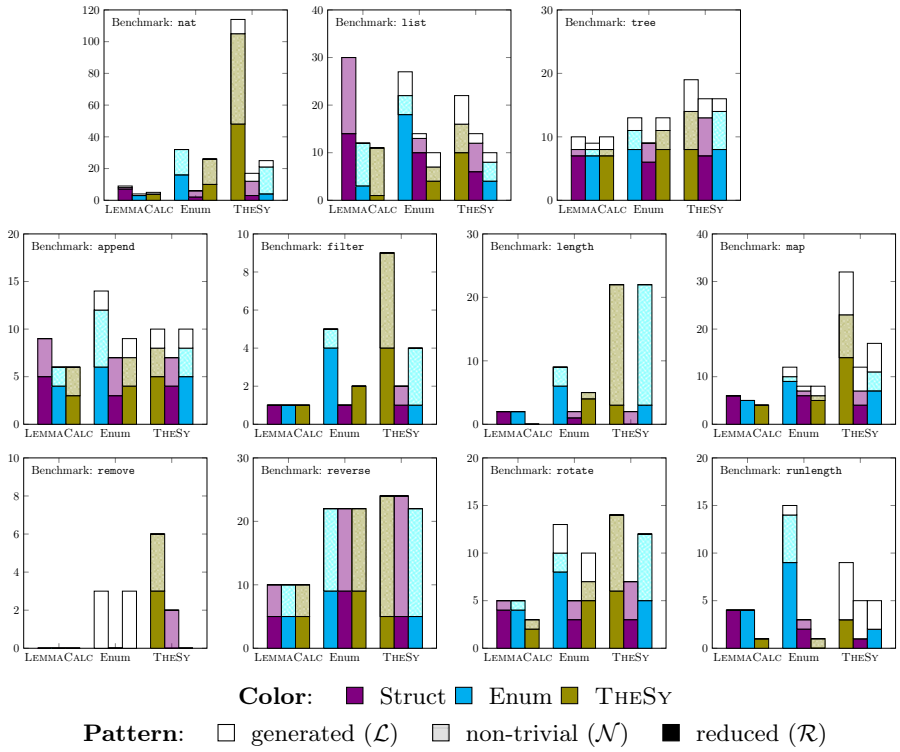

Fig. 1. Experiments for the full theories (top row) and individual benchmarks. The first bar in each group represents the number of lemmas found by the approach listed on the x-axis below (i.e., plain $\mathcal{L}$, $\mathcal{N}$, $\mathcal{R}$). The subsequent bars represent the proportion subsumed by other approaches (Def. 6), each again partitioned wrt. the classification. Four additional higher-order lemmas mentioning `map(succ, _)` are omitted from TheSy's result in `list` as they were not supported by the toolchain.

On benchmarks `length` and `reverse`, our methods miss out on lemmas for the functions `qlength` and `qreverse`, which both make use of an accumulator. As described in Ex. 8, it requires to find a nontrivial instance for the body of the synthetic function during accumulator removal, which our current implementation does not try, but which is generally in scope of the method. On both benchmarks, TheSy generates a lot of redundancy, because the functions involved can be related in many different ways (e.g., variations of Examples 1 and 2 and variants modulo properties of addition).

Regarding the full theories, benchmark `tree` is tractable for all methods, leading to similar results. Benchmark `nat` shows that the enumeration-based methods can make good use of their significantly longer running time (recall: many hours vs a few seconds for LemmaCalc). Specifically, our methods fail to produce most lemmas involving multiplication (such as distributivity wrt. addition, commutativity, associativity). Similarly, LemmaCalc does not discover

commutativity of addition—the algorithmic limitation is that the critical helper lemma $m + (n + 1) = (m + n) + 1$ is not calculated (it is too specific for our fusion algorithm), and the limitation of the implementation is that it enters a rewrite loop if this lemma is assumed as an additional fact in the theory.

RQ2: What is the impact of the size of the search space? Statistics on the size of the search space are given in Table 1. It varies not only with the number of functions in the theory, but also strongly depends on how many possible combinations there are. Of the candidates generated, a few hundreds are typically valid, and about 10% of those are of interest (i.e., need induction). The relatively low number of unknowns gives an upper bound on how many results were missed—by manual inspection most of these are in fact not valid.

LemmaCalc covers these theories very quickly, taking 1s–5s on all benchmarks except for list, where it takes 14s. By design, it scales more gracefully to larger theories (e.g., fusion takes square effort in the number of functions).

On the full theories, our baseline enumerator may take a significant amount of time. The exact time strongly depends on the effectiveness of the counterexample check. For example an earlier version of the implementation lacked support for conditional cases in functions, relying on the solver more heavily instead, so that run times on some benchmarks were tenfold to what is reported here. The timeout used for the solver plays an important factor, too. For example, the 1000ms per query in the experiment may be marginally better than 100ms only, and Z3 even exceeds this timeout significantly in many occasions (some queries time out after $\gg$10s only). Therefore the numbers shown should be a rough indication only what it takes to cover the search space.

We have aborted the run of TheSy after 26h resp. 21h on the `nat` and `list` benchmarks, and while TheSy terminates on some benchmarks within a minute, it stalls on others, e.g., on the `remove` benchmark, it produces lemmas until 13 minutes and then remains unproductive for many hours without any output (similar on `map` and `runlength`). Benchmark `remove` may suggest some internal implementation issue, that may have affected TheSy's performance on `list`.

Even if the run times shown are not to be taken as precise measures, it showcases that LemmaCalc in comparison to enumeration-based techniques is reliably quick, i.e., it can be incorporated into proof assistants and automated proof methods with little overhead while providing a similar benefit (cf. Fig. 1).

8 Related Work.

Calculational techniques for developing recursive functions go far back, notably to [7], which introduced the idea of unfold/fold transformations. An investigation of the theory of lists is provided by [2]. It already states many laws from a more general perspective. Follow-up work shows how to calculate such laws on pen and paper [3]. Program optimization using fusion-like techniques similarly have a long history [49,23] with various specialized approaches already developed, e.g., [48,50,53]. An approach that uses known lemmas to unblock fusion is discussed as

"warm-up rules" in [18]. General categorial notions that classify functions by their recursion schemes are based on a "zoo of morphisms" [32,24]. In comparison, [37] argues for a more direct approach that avoids fitting definitions into a particular shape, our algorithm in Sect. 4 is similar.

Fusion has been proposed as a building block for theorem proving before, e.g., [28,22,21]. Notably, Sonnex [45] demonstrates an effective implementation of these ideas and discusses many insights that underpin his procedure. In this work, the discovery of fold-functions—a limited form of synthesis—takes a similar role of accumulator removal in our work, in the sense that it unlocks proof steps that are out of scope of fusion. However, Sonnex does not consider theory exploration and its associated concerns in the absence of given proof goals as we do.

Accumulator transformations have been investigated in [16,27,30,16] with the goal of eliminating tail-recursion for ease of proof. Of these, the context manipulation techniques in [16] are very similar to our algorithm; our presentation is arguably more straight-forward and seems to encompass all four techniques mentioned. The deaccumulation technique in [17] employs a decomposition that is ultimately similar to the notion of "structured hylomorphisms" [25]. We leave it for future work to try these ideas.

Theory exploration has previously been approached by various techniques, e.g. [9,42], which utilize a conjecture generator based on testing and an induction principle enumerator. It constructs equations from a given set of functions and variables up to a certain depth, and a theorem prover is used at the backend to find the actually valid lemmas. Both approaches can be seen as instances of a more general approach called Syntax-Guided Synthesis (SyGuS) [1], that enjoys multiple applications in program verification and synthesis, [47,15,39] to name a few. A common drawback of these solutions is the exponentially-growing search space. RoughSpec [13] is an approach to overcome this problem by searching lemmas that fit particular patterns like distributivity. Another possibility to prune the search space is to rely on e-graphs [12,51], which is used by TheSy [42] and in also in Ruler [35]. FitSpec [4] represents an approach to filter redundant property-based tests, which could be applied to detect redundant conjectures.

When given a specific property to prove, theorem provers [26,8,46,52,54,43] are powered by various lemma discovery techniques that generate them by utilizing proof failures. Specifically, all of the above except [52,43] generalize a failure by replacing a common subterm by a fresh variable. AdtInd [52] instead uses syntax-guided enumeration [1] to enlarge and diversify the set of possible lemmas. Sivaraman et al. [43] uses a data-driven approach to finding lemmas: the goal itself gets an expression replaced by a hole. The synthesis specification is then formulated using input-output examples: valuations of the goal's variables are the inputs, and the valuations of the hole's original expressions are the outputs. Further data-driven approaches are [33,5] and the classic QuickCheck [44] which rely on testing only.

9 Conclusion

We have presented LemmaCalc, an approach for the synthesis of equational laws of recursive functions over algebraic data types. The approach is based on a novel combination of two program transformations. Key enabling factor is to integrate these in a procedure that chains facts discovered so far into the synthesis of subsequent lemmas. We have demonstrated that this approach to calculating lemmas is effective and efficient for many simple but non-trivial cases, and that it scales well to larger theories.

In contrast to enumeration-based methods, the lemmas generated by LemmaCalc, specifically with fusion and accumulator removal, often to match those a human engineer would specify by hand. Even though our calculations can work with intermediate synthetic functions that are not representable in the original theory, they rely on pre-existing building blocks (recursion structure, patterns, sub-expressions). This is key to taming the search space and also to limit redundancy in the discovered lemmas (cf. Sect. 7).

The lemmas discovered by approach in the evaluation and experiments are all good rewrite rules, i.e., we have not observed that they introduce cycles/matching loops (cf. Sect. 6), but we do not have a formal guarantee for this. We are not really sure yet how to define a more formal measure of utility of lemmas. Our intuition is that answering such a question leads to deep theoretical considerations such as working modulo some form of "canonicalization" (perhaps in the style of unified recursion schemes [24]), which gives a principled account of redundancy.

The approach presented is a suitable base for incorporating further transformations. While in its current form, LemmaCalc is somewhat restricted insofar that more complex lemmas than those shown in Sect. 7 are not necessarily in reach, which currently precludes us from conducting more "realistic" case studies. In the future, we want to investigate the potential of more elaborate decomposition of functions such as more principled approaches to function decomposition [25,17,30,27,24] to unlock more lemmas.

Finally, the current implementation is fairly robust but it also has some additional limitations beyond those mentioned in Sect. 3, such as not chaining on fusion with synthetic functions, not generating lemmas with duplicate variables, and not trying out more candidates during accumulator removal. Support for higher order is another feature of interest, as well as extending the approach to mutually recursive functions. Adding these involves significant additional engineering effort but it would enlarge the search space of LemmaCalc significantly, leading to larger execution times, too. As an outlook, it appears promising to combine enumeration with calculational techniques in a principled way to leverage the respective strengths.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-Guided Synthesis. In: FMCAD. pp. 1–17. IEEE (2013)

2. Bird, R.S.: An introduction to the theory of lists. Springer (1987)
3. Bird, R.S.: Algebraic identities for program calculation. The Computer Journal **32**(2), 122–126 (1989)
4. Braquehais, R., Runciman, C.: Fitspec: refining property sets for functional testing. In: Proceedings of the 9th International Symposium on Haskell. pp. 1–12 (2016)
5. Braquehais, R., Runciman, C.: Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. pp. 40–51 (2017)
6. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., Smaill, A.: Rippling: A heuristic for guiding inductive proofs. Artificial Intelligence **62**(2), 185 – 253 (1993)
7. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM (JACM) **24**(1), 44–67 (1977)
8. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The ACL2 Sedan Theorem Proving System. In: TACAS. pp. 291–295. LNCS, Springer (2011)
9. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) CADE. Lecture Notes in Computer Science, vol. 7898, pp. 392–406. Springer (2013)
10. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Removing algebraic data types from constrained horn clauses using difference predicates. In: IJCAI. Lecture Notes in Computer Science, vol. 12166, pp. 83–102. Springer (2020)
11. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: Formal models and semantics, pp. 243–320. Elsevier (1990)
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005). https://doi.org/10.1145/1066100.1066102, https://doi.org/10.1145/1066100.1066102
13. Einarsdóttir, S.H., Smallbone, N., Johansson, M.: Template-based theory exploration: discovering properties of functional programs by testing. In: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages. pp. 67–78 (2020)
14. Fedyukovich, G., Ernst, G.: Bridging arrays and ADTs in recursive proofs. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 24–42. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_2, https://doi.org/10.1007/978-3-030-72013-1_2
15. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling Invariants from Frequency Distributions. In: FMCAD. pp. 100–107. IEEE (2017)
16. Giesl, J.: Context-moving transformations for function verification. In: LOPSTR. pp. 293–312. Springer (1999)
17. Giesl, J., Kühnemann, A., Voigtländer, J.: Deaccumulation techniques for improving provability. The Journal of Logic and Algebraic Programming **71**(2), 79–113 (2007)
18. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proceedings of the conference on Functional programming languages and computer architecture. pp. 223–232 (1993)
19. Govind, H., Shoham, S., Gurfinkel, A.: Solving constrained horn clauses modulo algebraic data types and recursive functions. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498722, https://doi.org/10.1145/3498722

20. Hajdú, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: FMCAD. pp. 1–10. IEEE (2021)
21. Hamilton, G.W.: Poítin: Distilling theorems from conjectures. Electronic Notes in Theoretical Computer Science **151**(1), 143–160 (2006)
22. Hamilton, G.W.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 61–70 (2007)
23. Hinze, R., Harper, T., James, D.W.: Theory and practice of fusion. In: Symposium on Implementation and Application of Functional Languages. pp. 19–37. Springer (2010)
24. Hinze, R., Wu, N., Gibbons, J.: Unifying structured recursion schemes. ACM SIGPLAN Notices **48**(9), 209–220 (2013)
25. Hu, Z., Iwasaki, H., Takeichi, M.: Deriving structural hylomorphisms from recursive definitions. ACM Sigplan Notices **31**(6), 73–82 (1996)
26. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 291–306. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
27. Kapur, D., Subramaniam, M.: Automatic generation of simple lemmas from recursive definitions using decision procedures–preliminary report–. In: Advances in Computing Science–ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation: 8th Asian Computing Science Conference, Mumbai, India, December 10-12, 2003. Proceedings 8. pp. 125–145. Springer (2003)
28. Klyuchnikov, I.G., Romanenko, S.A.: Proving the equivalence of higher-order terms by means of supercompilation. In: Ershov Memorial Conference. pp. 193–205. Springer (2009)
29. Kostyukov, Y., Mordvinov, D., Fedyukovich, G.: Beyond the Elementary Representations of Program Invariants Over Algebraic Data Types. In: PLDI. pp. 451–465 (2021)
30. Kühnemann, A., Glück, R., Kakehi, K.: Relating accumulative and non-accumulative functional programs. In: RTA. vol. 1, pp. 154–168. Springer (2001)
31. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR. LNCS, vol. 6355, pp. 348–370. Springer (2010)
32. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: FPCA. vol. 91, pp. 124–144 (1991)
33. Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: PLDI. pp. 1–15 (2020)
34. Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
35. Nandi, C., Willsey, M., Zhu, A., Wang, Y.R., Saiki, B., Anderson, A., Schulz, A., Grossman, D., Tatlock, Z.: Rewrite rule inference using equality saturation. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–28 (2021)
36. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
37. Ohori, A., Sasano, I.: Lightweight fusion by fixed point promotion. ACM SIGPLAN Notices **42**(1), 143–154 (2007)
38. Pham, T., Gacek, A., Whalen, M.W.: Reasoning about algebraic data types with abstractions. J. Autom. Reason. **57**(4), 281–318 (2016)
39. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In: CAV, Part II. LNCS, vol. 11562, pp. 74–83. Springer (2019)

40. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: VMCAI. LNCS, vol. 8931, pp. 80–98. Springer (2015)
41. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM (JACM) **12**(1), 23–41 (1965)
42. Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 125–148. Springer (2021)
43. Sivaraman, A., Sanchez-Stern, A., Chen, B., Lerner, S., Millstein, T.D.: Data-driven lemma synthesis for interactive proofs. Proc. ACM Program. Lang. **6**(OOPSLA2), 505–531 (2022)
44. Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. Journal of Functional Programming **27**, e18 (2017)
45. Sonnex, W.: Fixed point promotion: taking the induction out of automated induction. Tech. rep., University of Cambridge, Computer Laboratory (2017)
46. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: TACAS. LNCS, vol. 7214, pp. 407–421. Springer (2012)
47. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI. pp. 223–234. ACM (2009)
48. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Proceedings of the seventh international conference on Functional programming languages and computer architecture. pp. 306–313 (1995)
49. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(3), 292–325 (1986)
50. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: ESOP'88: 2nd European Symposium on Programming Nancy, France, March 21–24, 1988 Proceedings. pp. 344–358. Springer (2005)
51. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proceedings of the ACM on Programming Languages **5**(POPL), 1–29 (2021)
52. Yang, W., Fedyukovich, G., Gupta, A.: Lemma Synthesis for Automating Induction over Algebraic Data Types. In: CP. LNCS, vol. 11802, pp. 600–617. Springer (2019)
53. Yokoyama, T., Hu, Z., Takeichi, M.: Calculation rules for warming-up in fusion transformation. In: the 2005 Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia. pp. 399–412. Citeseer (2005)
54. Zavalía, L., Chernigovskaia, L., Fedyukovich, G.: Solving constrained horn clauses over algebraic data types. In: Dragoi, C., Emmi, M., Wang, J. (eds.) Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13881, pp. 341–365. Springer (2023). https://doi.org/10.1007/978-3-031-24950-1_16, https://doi.org/10.1007/978-3-031-24950-1_16

A Further Examples

We show two further examples, Ex. 7 for fusion, and Ex. 8 for accumulator removal.

Example 7 (Binary Trees). The data type for binary trees over type `Elem` is

$$\textbf{data}\ \texttt{Tree} = \texttt{leaf} \mid \texttt{node}(\texttt{left} : \texttt{Tree}, \texttt{value} : \texttt{Elem}, \texttt{right} : \texttt{Tree})$$

Function `elems` over binary trees computes a list containing its elements by pre-order traversal and function $\texttt{size}(t)$ counts the number of nodes.

$$\texttt{size}(\texttt{leaf}) \coloneqq 0 \qquad \texttt{size}(\texttt{node}(l, x, r)) \coloneqq \texttt{size}(l) + \texttt{size}(r) + 1 \tag{10}$$

$$\texttt{elems}(\texttt{leaf}) \coloneqq [] \qquad \texttt{elems}(\texttt{node}(l, x, r)) \coloneqq x :: (\texttt{elems}(l) \mathbin{+\!\!+} \texttt{elems}(r)) \tag{11}$$

Goal is to fuse `length_elems` with $\texttt{length\_elems}(t) = \texttt{length}(\texttt{elems}(t))$, expecting that it will turn out to be equivalent to `size`.

We take apart the two cases in (11) corresponding to line 2 in Alg. 1. For the base case, $e_0^g = []$ and $\Gamma \vdash \texttt{length}([]) \rightsquigarrow 0$ in line 4 of Alg. 1 by the definition of `length` that is part of Γ, so that $\texttt{length\_elems}(\texttt{leaf}) \coloneqq 0$.

In the recursive case, $e_1^g = e' = x :: (\texttt{elems}(l) \mathbin{+\!\!+} \texttt{elems}(r))$. Pattern match of the base case of `length` is refuted by $[] \bot e'$ (line 10 of Alg. 1). For pattern $y :: ys$ of the recursive case of `length` we get a unifier σ with $\sigma(y) = x$ and $\sigma(ys) = \texttt{elems}(l) \mathbin{+\!\!+} \texttt{elems}(r)$. We then look at $\sigma(\texttt{length}(ys) + 1) = \texttt{length}(\texttt{elems}(l) \mathbin{+\!\!+} \texttt{elems}(r)) + 1$. It is not possible to *immediately* apply the fold rule that collapses occurrences of $\texttt{length}(\texttt{elems}(\_))$ because of the intermediate occurrence of function `++`. Instead, we need lemma (4) to unblock the situation which is done by the second use of rewriting in line 9 as $\texttt{length}(\texttt{elems}(l) \mathbin{+\!\!+} \texttt{elems}(r)) + 1 \rightsquigarrow e''$ with $e'' = \texttt{length\_elems}(l)) + \texttt{length\_elems}(r)) + 1$, which is now in fused form and can be used as the right-hand side of the case $\texttt{length\_elems}(\texttt{node}(l, x, r)) \coloneqq \texttt{length\_elems}(l) + \texttt{length\_elems}(r) + 1$. This indeed gives us a definition that is equivalent to `size`. From this, we can extract $\texttt{length}(\texttt{elems}(t)) = \texttt{size}(t)$. ■

Example 8 (Reversing Lists using an Accumulator). Function `reverse` is inefficient (quadratic runtime). Function `qreverse`, defined below, avoids this by introducing an accumulator *us*

$$\texttt{qreverse}([], us) = us \qquad \texttt{qreverse}(x :: xs, us) = \texttt{qreverse}(xs, x :: us)$$

Algorithm RemoveAcc for the accumulator *us* calculates

$$\texttt{qreverse}'([]) = b_1' \qquad \texttt{qreverse}'(x :: xs) = b_2'(x, xs, \texttt{qreverse}'(xs))$$

The solution is $b_1 = []$ as right-neutral of `++`, as well as $b_2'(x, xs, ys) = ys \mathbin{+\!\!+} (x :: [])$, which produces $\texttt{qreverse}' = \texttt{reverse}$, i.e., removing the accumulator from `qreverse` yields again the more inefficient version so that $\texttt{qreverse}(xs, us) = \texttt{reverse}(xs) \mathbin{+\!\!+} us$. Note that the choice b_2' is *not* the canonical choice, as we have $b_2(ys) = ys$ from `reverse` in line 10 of Alg. 2; our implementation therefore currently misses this result. ■

B Soundness Proofs

We are working with typed functions $f : t_1, \ldots, t_n \to t$, defined by cases

$$\textbf{function definition} \qquad f(\overline{p}_1) \coloneqq e_1\ \texttt{if}\ \varphi_1 \quad \cdots \quad f(\overline{p}_m) \coloneqq e_m\ \texttt{if}\ \varphi_m$$

Recall the assumptions placed on function definitions namely that functions terminate, and that cases match are mutually disjoint and together complete. We formalize these conditions below and prove that the synthetic functions produced by the transformations preserve these. Furthermore, we prove that all generated lemmas are valid.

Definition 7 (Termination). *A function f is terminating if and only if there is a corresponding well-founded order $\prec_f$ that connects arguments to recursive calls, i.e., for each recursive case $f(\overline{p}_i) \coloneqq e_i\big(\, f(\overline{e}) \,\big)$ $\texttt{if}$ φ_i of the definition of f satisfies $\forall \overline{x}.\ \overline{e} \prec_f \overline{p}_i$ where $\overline{x} = \mathit{free}(\overline{p}_i)$ are the variables in scope.*

Definition 8. *Let $V_t = \{v \mid v\colon t\}$ be the carrier set of values v of type t.*

We fix a set of definitions Δ. We write $\Delta \models \varphi$ or just φ *holds* if formula φ semantically follows from Δ.

Definition 9. *Let $[\![\overline{p}\ \texttt{if}\ \varphi]\!] = \{\ \overline{v} \mid \exists\ \sigma.\ \sigma(\overline{p}) = \overline{v} \wedge \Delta \models \sigma(\varphi)\ \}$ be the set of values $\overline{v}$ that match pattern $\overline{p}$ via some substitution σ so that the guard φ holds.*

Definition 10 (Pattern Disjointness). *The patterns $\overline{p}_1, \ldots, \overline{p}_m$ of function f are disjoint if $[\![\overline{p}_i\ \texttt{if}\ \varphi_i]\!] \cap [\![\overline{p}_j\ \texttt{if}\ \varphi_j]\!] = \varnothing$ for all $1 \leq i < j < m$.*

Definition 11 (Pattern Completeness). *The patterns $\overline{p}_1, \ldots, \overline{p}_m$ of function f are complete if $V_{t_1} \times \cdots \times V_{t_n} = \bigcup_{i=1,\ldots,m} [\![\overline{p}_i\ \texttt{if}\ \varphi_i]\!]$.*

We remark that the $\supseteq$ direction always holds by type-correctness, therefore it will be sufficient to demonstrate the $\subseteq$ direction.

Lemma 2. $(A_1 \cap A_2) \times (B_1 \cap B_2) = (A_1 \times B_1) \cap (A_2 \times B_2)$.

Lemma 3. *For $A_1 \subseteq A_2$ and $B_1 \subseteq B_2$ we have $A_2 \cap B_2 = \varnothing \implies A_1 \cap B_1 = \varnothing$.*

Lemma 4. *If $\mathit{free}(\varphi) \subseteq \mathit{free}(\overline{p})$ and $\mathit{free}(\psi) \subseteq \mathit{free}(\overline{q})$ then matches of p, q over disjoint variables $\mathit{free}(p) \cap \mathit{free}(q) = \varnothing$ can be split into a cross-product for the individual matches $[\![\overline{p}, \overline{q}\ \texttt{if}\ \varphi \wedge \psi]\!] = [\![\overline{p}\ \texttt{if}\ \varphi]\!] \times [\![\overline{q}\ \texttt{if}\ \psi]\!]$.*

Lemma 5. *Substitution narrows down pattern matches $[\![\sigma(p)\ \texttt{if}\ \sigma(\varphi)]\!] \subseteq [\![p\ \texttt{if}\ \varphi]\!]$.*

Lemma 6 (f-Induction). *With a well-founded order $\prec_f$ of a terminating function $f\colon \overline{t} \to t$ that satisfies Def. 11 we can prove any property $P(\overline{z})$ over $\overline{z}\colon \overline{t}$ by induction:*[5]

$$\left(\bigwedge_{i=1,\ldots,m} \forall\ \overline{x}.\ P(\overline{e}) \wedge \varphi_i \implies P(\overline{p}_i) \right) \implies \big(\forall\ \overline{z}.\ P(\overline{z})\big)$$

[5] `(induction` $\overline{z}$ `rule:` f`.induct)` in Isabelle/HOL.

B.1 Properties of Fuse (Alg. 1 in Sect. 4)

Lemma 7 (Termination of fg). *All recursive fg calls are introduced by a fold rule $\overline{y} \prec_g p_j^g \implies f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y})$ (line 3). Therefore, $\prec_{fg} = \prec_g$ witnesses termination of fg.* □

Lemma 8 (Pattern Disjointness of fg). *We prove disjointness of two cases $(i,j) \neq (i',j')$ both generated by line 9, the other combinations wrt. line 4 are analogous. If $i \neq i'$, then from pattern disjointness of f*

$$[\![\overline{p}_i^f \,\texttt{if}\, \varphi_i^f]\!] \cap [\![\overline{p}_{i'}^f \,\texttt{if}\, \varphi_{i'}^f]\!] = \varnothing$$

$$\overset{\varnothing \times A = \varnothing}{\Longrightarrow}$$

$$([\![\overline{p}_i^f \,\texttt{if}\, \varphi_i^f]\!] \cap [\![\overline{p}_{i'}^f \,\texttt{if}\, \varphi_{i'}^f]\!]) \times ([\![\overline{p}_j^g \,\texttt{if}\, \varphi_{j'}^g]\!] \cap [\![\overline{p}_j^g \,\texttt{if}\, \varphi_{j'}^g]\!]) = \varnothing$$

$$\overset{\text{Lemma 2}}{\Longleftrightarrow}$$

$$([\![\overline{p}_i^f \,\texttt{if}\, \varphi_i^f]\!] \times [\![\overline{p}_j^g \,\texttt{if}\, \varphi_{j'}^g]\!]) \cap ([\![\overline{p}_{i'}^f \,\texttt{if}\, \varphi_{i'}^f]\!] \times [\![\overline{p}_j^g \,\texttt{if}\, \varphi_{j'}^g]\!]) = \varnothing$$

$$\overset{\text{Lemma 4}}{\Longleftrightarrow}$$

$$[\![\overline{p}_i^f, \overline{p}_j^g \;\texttt{if}\; \varphi_i^f \wedge \varphi_j^g]\!] \cap [\![\overline{p}_{i'}^f, \overline{p}_{j'}^g \;\texttt{if}\; \varphi_{i'}^f \wedge \varphi_{j'}^g]\!] = \varnothing$$

$$\overset{\text{Lemmas 3 and 5}}{\Longrightarrow}$$

$$[\![\sigma(\overline{p}_i^f, \overline{p}_j^g) \;\texttt{if}\; \sigma(\varphi_i^f \wedge \varphi_j^g)]\!] \cap [\![\sigma(\overline{p}_{i'}^f, \overline{p}_{j'}^g) \;\texttt{if}\; \sigma(\varphi_{i'}^f \wedge \varphi_{j'}^g)]\!] = \varnothing$$

The argument for $i = i'$ and $j \neq j'$ is analogous via pattern disjointness of g. □

Lemma 9. *If $e = e'$ and $v = \sigma(e)$ then $v = \sigma(e')$.*

Proof. Congruence of substitution with respect to semantic equality $e = e'$.

Lemma 10 (Pattern Completeness of fg). *Let $g \colon \overline{t}^g \to t$ and $f \colon \overline{t}^f, t_g \to t'$, and assume that fusion successfully computed a definition of fg. We prove pattern completeness of fg.*

Proof. in the light of the remark below Def. 11 it suffices that each arbitrary $\overline{v} \in V_{\overline{t}^f}$ and $\overline{w} \in V_{\overline{t}^g}$ is covered by some case in set Δ returned by Alg. 1.

By pattern completeness of g, there is a case j of g with $\overline{w} \in [\![\overline{p}_j^g \;\texttt{if}\; \varphi_j^g]\!]$, and by Def. 9 there is a substitution τ^g over $\mathit{free}(\overline{p}_j^g)$ with

$$\tau^g(\overline{p}_j^g) = \overline{w} \quad \text{and} \quad \tau^g(\varphi_j^g) \text{ holds} \tag{12}$$

If case j of g can be processed by line 4, we have $[\![\overline{x}, \overline{p}_j^g \;\texttt{if}\; \varphi_j^g]\!] = V_{\overline{t}^f} \times [\![\overline{p}_j^g \;\texttt{if}\; \varphi_j^g]\!]$. Otherwise, by pattern completeness of f, there is a case i of f that matches the result e_j^g of g instantiated with τ^g, i.e., $\overline{v}, \tau^g(e_j^g) \in [\![\overline{p}_i^f, q^f \;\texttt{if}\; \varphi_i]\!]$ and by Def. 9 there is a substitution τ^f over $\mathit{free}(\overline{p}_j^f, q^f)$ with

$$\tau^f(\overline{p}_i^f, q_i^f) = \overline{v}, \tau^g(e_j^g) \quad \text{and} \quad \tau^f(\varphi_i^f) \text{ holds} \tag{13}$$

Note, τ^f and τ^g are over disjoint variables by the condition in line 8 of Alg. 1. Therefore, we can freely switch between $\tau = (\tau^f \cup \tau^g)$ and the more specific substitutions for expressions over variables of either f or g exclusively. In particular both (12) and (13) hold for τ, too, and we have $\tau(q_i^f) = \tau(e_j^g)$.

At this point we have to justify that we actually satisfy the test in line 9, but it is the only possibility: Having a unifier τ contradicts the test for refutation in line 12 and having fused fg successfully in the first place rules out line 16. Therefore, there exists the consituents of line 9, in particular e' with $e' = e_j^g$ (by soundness of rewriting) and the most general unifier σ. Def. 3 splits $\tau = \tau' \circ \sigma$ for some τ', which can be partitioned into the respective sets of variables again, so that $\tau^f = \tau'_f \circ \sigma$ and so that $\tau^g = \tau'_g \circ \sigma$.

It remains to be shown that $\overline{v}, \overline{w} \in [\![\overline{p} \texttt{ if } \varphi]\!]$ for $\overline{p}$ and φ constructed by line 10. The substitution that witnesses Def. 9 is given as τ':

$$\begin{aligned}\tau'(\overline{p}) &= \tau'(\sigma(\overline{p}_i^f, \overline{p}_j^g)) \\ &= \tau'(\sigma(\overline{p}_i^f), \tau'(\sigma(\overline{p}_j^g)) \\ &= \tau'_f(\sigma(\overline{p}_i^f), \tau'_g(\sigma(\overline{p}_j^g)) = \overline{v}, \overline{w}\end{aligned}$$

The reasoning for the guard is analogous.

Lemma 11. *Fusion lemma $fg(\overline{x}, \overline{y}) = f(\overline{x}, g(\overline{y}))$ holds.*

Proof. By induction over $\prec_{fg}$ (cf. Lemma 6 via Lemma 7) and by taking apart the definitional cases of fg. Note that we may assume the respective guard of fg. By equational reasoning and assuming φ_j^g, for line 4 we have

$$fg(\overline{x}, \overline{p}_j^g) \overset{\text{def. } fg}{=} e' \overset{\Gamma_{fg} \text{ valid}}{=} f(\overline{x}, e_j^g) \overset{\text{def. } g}{=} f(\overline{x}, g(\overline{p}_j^g))$$

For line 9, assuming $\sigma(\varphi_i^f \wedge \varphi_j^g)$ we have

$$\begin{aligned} fg(\sigma(\overline{p}_i^f, \overline{p}_j^g)) &\overset{\text{def. } fg}{=} e' \overset{\Gamma_{fg} \text{ valid}}{=} \sigma(e_i^f) \\ &\overset{\text{def. } f}{=} f(\sigma(\overline{p}_i^f), \sigma(q_i^f)) \overset{\text{def. } 3}{=} f(\sigma(\overline{p}_i^f), e') \\ &\overset{\Gamma \text{ valid}}{=} f(\sigma(\overline{p}_i^f), e_j^g) \overset{\text{def. } g}{=} f(\sigma(\overline{p}_i^f), g(\sigma(\overline{p}_j^g))) \end{aligned}$$

Steps justified by validity of Γ rely on rewriting to produce valid equations because Γ contains definitions and valid lemmas only. Γ_{fg} is valid, because its additional rule is just the inductive hypothesis. The step marked def. 3 holds because unification produces syntactically identical expressions. Steps by the respective definitions of f and g specialize the respective pattern variables, and of course one has to ensure that the respective guard follows from that of fg.

B.2 Properties of RemoveAcc (Alg. 2 in Sect. 5)

Lemma 12 (Termination). *f' terminates by the well-founded order $\prec_{f'}$ defined as the least fixpoint of the set of implications over all i, j, where i indexes defining cases and j indexes recursive calls of that case*

$$\forall\, \overline{x}.\ \left(\forall\, u.\ \overline{e}, a_i^j(u) \prec_{f'} \overline{p}_i, u\right) \implies \overline{e} \prec_{f'} \overline{p}_i$$

Proof. where $\overline{x} = \mathit{free}(\overline{p}_i)$. Since $\prec_{f'}$ is a least fixpoint, it is well-founded. It remains to show that it covers all recursive calls in f', which is apparent from the construction.

Lemma 13 (Pattern Completeness and Disjointness). *Because the accumulator $u\colon t_u$ is always matched as a variable and because it cannot occur in guards φ_i we have $[\![\overline{p}_i, u \texttt{ if } \varphi_i]\!] = [\![\overline{p}_i \texttt{ if } \varphi_i]\!] \times V_{t_u}$ from Lemmas 2 and 4.* □

Lemma 14. *Lemma $f(\overline{x}, u) = e^?(f'(\overline{x}), \overline{x}', u)$ holds.*

Proof. By f-induction over $\prec_f$ (cf: Lemma 6). The base case follows from the condition in line 7. Instantiating the condition in line 13 with $\overline{y} = f(y^1), \ldots, f(y^k)$ proves the correspondence in the recursive case by the inductive hypothesis.

C Benchmarks

Table 1. Statistics on benchmark theories used in the comparison. Here $|F|$ is the number of functions. The number of candidates is $\Sigma_{f(\overline{x},g(\overline{y}))}|S_3^2(\overline{x}, \overline{y}, t_f)|$, where t_f is the respective result type of f, i.e., potential right-hand sides to equations (7) of depth $d = 3$ and max $o = 2$ occurrences of each variable. true: proved by Z3 from axioms/prior lemmas, $|\Lambda|$: lemmas proved by induction+Z3 (i.e., including background lemmas), **?**: candidates with unknown status. Time is shown in hours:minutes:seconds. For TheSy, we report the time of the last lemma found. For those benchmarks on which the tool did not terminate, we give a rough indication when it was cancelled. Note, except for `nat`, TheSy stopped reporting lemmas long before. †: Interrupted during the second round of lemma checking when the backend solver got stuck without honoring the timeout per query. **In comparison, LemmaCalc takes 14s on `list` and 1s–5s on all other benchmarks.**

		baseline enumerator statistics					TheSy	
benchmark	$\|F\|$	candidates	true	$\|\Lambda\|$	**?**	time	last	killed
`nat`	8	1 131 799	501	32	1759	6:50:00	26:38:14	>26h
`list`	18	319 019	408	32	522	1:48:22	10:55:14	>21h
`tree`	11	123 178	130	20	38	11:25	16:47	
`append`	5	15 058	133	22	5	02:03	04:32	
`filter`	6	398	2	5	16	02:11	00:02	
`length`	5	7 066	558	12	1	01:59	00:00	
`map`	8	34 726	103	13	35	07:18	37:33	>11h
`remove`	7	32 302	117	14	13	22:11	13:01	>11h
`reverse`	4	127 926	427	22	1	03:29	00:02	
`rotate`	6	12 784	124	20	43	08:50	6:54:22	>11h
`runlength`	7	68 311	182	23	847	†1:12:12	00:40	>11h

The additional functions present in the respective benchmarks are listed below:

- append: add, snoc, ++, length, count
- filter: not, length, filter, all, ex, countif
- length: length, length°, qlength(tail-recursive)
- map: leq, lt, length, map, take, drop
- remove: not, add, sub, length, contains, remove, count
- reverse: reverse, reverse°, qreverse(tail-recursive)
- rotate: leq, add, append, length, reverse, rotate
- runlength: add, mul, ++, sum, sumruns, decode, is_runs

Function not is the logical negation. Functions/predicates add, sub, mul, leq, lt are structurally recursive definitions over natural numbers for $+$, $-$, $*$, $\leq$, and $<$. Function filter keeps elements that satisfy a given predicate, countif counts them, and all/ex test if all/some element satisfies a given predicate. Function rotate reverses a prefix of a given list. Benchmark runlength implements the decoder for a sequence of runs, given as a pair of lists that record elements resp. the number of their occurrence in a run.[6] A critical lemma connects sum over the decoded sequence to sumruns that works on the coded one.

[6] https://en.wikipedia.org/wiki/Run-length_encoding