Quick Theory Exploration for Algebraic Data Types via Program Transformations

Gidon Ernst, Grigory Fedyukovich, and Robin Sögtrop

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/deforestation and accomulator removal). Being calculational, our approach avoids the 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 the different methods have complementary strengths and that each can produce lemmas not found by the other. We demonstrate that LemmaCalc scales to larger theories with a small fraction of the cost of enumeration.

1 Introduction

Algebraic Data Types (ADTs) enable formal modeling of programs in proof assistants and verification tools, e.g. [52,54,18,29,12,32,9,37,20,39,31,26]. They leverage equational reasoning and induction when proving properties about *recursively-defined functions* over ADTs, often relying on a set of lemmas that explain possible interactions and relationships among these functions. The problem of discovering of such lemmas is sometimes called "*theory exploration*", and it often requires either a manual effort of a proof engineer, or an exhaustive search (a.k.a. enumeration) over millions of candidate formulas [7,43,1]. The key feature of enumeration is that in principle it can cover the entire search space and is therefore complete relative to the proof oracle used. Yet, a vast majority of these candidates would be wrong or not useful, and the search space grows exponentially with the number of function symbols. This can be very expensive, especially given that libraries of interactive theorem provers like Isabelle/HOL, Coq, or Lean contain many dozens of existing function definitions that may be relevant to a particular use case for theory exploration; in practice performance of automated provers may also degrade when given large sets of definitions. We are interested in approaches that can scale much better to such scenarios.

We present LemmaCalc, a new, fully automated, approach to theory exploration that avoids enumerating large amounts of formulas and minimizes the reliance on automated (inductive) theorem provers to validate them. The key

difference is that LemmaCalc leverages *calculational techniques*, based on unfold/fold transformations [4] of recursive functions. This idea, originally developed in the context of program optimization (e.g. "supercompilation" [49]), has been recognized to be a useful inductive *proof* method [45,28,19], too.

At a high-level, our approach takes a set of functions, applies a series of interleaving fusion and accumulator transformations. Each of them rearranges a given computation into new synthetic functions. LemmaCalc then relates them with each other and with the original functions. Key to effectiveness is the *combination* of the transformations, which to the best of our knowledge is novel. Specifically, for this work we rely on *fusion* [33] (resp. supercompilation [49], deforestation [50]), *accumulator removal* by a technique similar to context shift [15], and a novel technique for the *inference of recursive preconditions*. While we have found this combination to be effective, our approach is generic and could be extended to cover other transformations like [16,30,27].

Our fully automatic implementation has been evaluated in three theories within ADTs: Peano natural numbers, lists, and trees. 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 [43] would such theories in *several hours*, LemmaCalc consistently does it in a few seconds. Regarding strengths of lemmas found, there is high variability and no single approach is best: The proportion of lemmas generated by one method is implied by those discovered by another between $\sim$10% to 100%.

Contribution and Structure. In this paper, we present LemmaCalc, an approach that leverages program transformation techniques for *theory exploration*. In section 2 we motivate how the *combination* of techniques is important. We describe how to adapt two known transformations, fusion and accumulator removal, to the purpose of lemma discovery (sections 4 and 5) and present a novel approach to the inference of preconditions in section 6. We have implemented LemmaCalc and evaluated it on theories over natural numbers, lists, and trees (sections 7 and 8): LemmaCalc is similarly effective but its performance is orders of magnitude faster when compared to enumeration-based techniques.

Related Work. Calculational techniques for developing recursive functions go far back, notably to [4], 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 [17]. General categorial notions that classify functions by their recursion schemes are based on a "zoo of morphisms" [33,24]. In comparison, [36] argues for a more direct approach that avoids fitting definitions into a particular shape, our algorithm in section 4 is similar. These ideas have been used for theorem proving, e.g., [28,22,21,45]

Techniques for deaccumulation have been investigated for verification in [15,27,30,15] with the goal of switching to functions that are not tail-recursive for ease of proof. Of these, the context manipulation techniques in [15] 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 [16] 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 have previously been approached by various techniques, e.g. [7,43], 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. A common drawback of these solutions is the exponentially-growing search space. These approaches need powerful and/or aggressive heuristics to prune the space substantially such as e-graphs [11,51].

When given a specific property to prove, theorem provers [26,5,46,52,54,44] are powered by various lemma discovery techniques that generate them by utilizing proof failures. Specifically, all of the above except [52,44] 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. [44] 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. Another data-driven by Miltner et al. [34] justifies the found invariants by testing only, which bypasses the need for inductive proofs.

2 Motivation and Overview

At a high-level, our approach takes a set of functions $\{f, g, \dots\}$, applies a series of fusion and accumulator removal steps 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[1]

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

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

where fg and f' are synthetic recursive functions, and $e^{?}$ is instantiated as an expression. 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:

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

[1] 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.

We also generalize the shape of equations by including a synthetic *precondition* *pre* which defines a scope where the corresponding lemma is applicable:

$$\textbf{conditional replacement:} \quad pre(\overline{x}) \implies f(\overline{x}) \in \{x_i, c(\overline{x}')\} \tag{4}$$

$$\textbf{conditional equality:} \quad pre(\overline{x}, \overline{y}, \overline{z}) \implies f(\overline{x}, \overline{y}) = h(\pi(\overline{x}, \overline{z})) \tag{5}$$

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"), `length`, and $\texttt{length}^\circ$ (a generalization of `length`) defined in the next three rows, respectively:

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

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

$$\texttt{length}^\circ(\texttt{[]}, u) \coloneqq u \qquad \texttt{length}^\circ(x :: xs, u) \coloneqq \texttt{length}^\circ(xs, u) + 1$$

Our approach calculates the lemmas shown below, among others, each of which would be validated by automated induction provers:

$$\texttt{length}(xs \mathbin{+\!\!+} ys) = \texttt{length}(xs) + \texttt{length}(ys) \tag{6}$$

$$u = 0 \implies \texttt{length}^\circ(xs, u) = \texttt{length}(xs) \tag{7}$$

In the rest of the section, we illustrate that it is critical to *combine* the respective transformations in LemmaCalc to leverage their full potential. We keep the examples simple, but refer the reader to appendix C for more detailed ones.

The first transformation, **fixpoint 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 $g = \texttt{++}$ produces the following definition for synthetic function $fg = \texttt{length}_{\texttt{++}}$

$$\texttt{length}_{\texttt{++}}(\texttt{[]}, \underline{ys}) \coloneqq \underline{\texttt{length}(ys)}$$

$$\texttt{length}_{\texttt{++}}(x :: xs, \underline{ys}) \coloneqq \texttt{length}_{\texttt{++}}(xs, \underline{ys}) + 1$$

that satisfies $\texttt{length}(xs \mathbin{+\!\!+} ys) = \texttt{length}_{\texttt{++}}(xs, ys)$ by construction. The difference in definitions of $\texttt{length}_{\texttt{++}}$ and `length` is underlined: the entire base case and additional parameter in the inductive case. ■

Fusion tends to regularize the way in which computations are laid out. Complementary, **accumulator removal**, untangles computations again in a differently form, by relating functions with accumulators and those without (we regard ys in example 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 outpus 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 example 1:

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

$$\texttt{length}'_{+\!+}([\,]) \coloneqq 0 \tag{8}$$

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

$$\text{so that} \quad \texttt{length}_{+\!+}(xs, ys) = \texttt{length}'_{+\!+}(xs) + \underline{\texttt{length}(ys)} \quad \text{holds,} \tag{9}$$

where the underlined part is the base case expression from example 1 and $e^?$ in (2) is instantiated by $+$. This solution can be found algorithmically as described in section 5 by relying on neutral elements like 0 to replace base case expressions like in (8) and use the respective operator like $+$ in the solution for $e^?$. Then, $\texttt{length}'_{+\!+}$ is structurally equivalent to $\texttt{length}$ and we can apply the corresponding **replacement** lemma (3) to (9) to conclude (6). ■

Finally, **conditional equations** can be discovered by algorithmically abducing (intuitively, finding preconditions) recursive predicates that enforce desired equations of form (4) and (5).

Example 3. We synthesize pre with $pre(xs, u) \implies \texttt{length}^\circ(xs, u) = \texttt{length}(xs)$ by comparing the respective cases in the definition of $\texttt{length}^\circ$ to those of $\texttt{length}$ with respect to common arguments xs. Predicate pre inherits the recursive traversal over that shared argument:

$$\left.\begin{aligned} \texttt{length}^\circ([\,], u) &\coloneqq u \\ \texttt{length}([\,]) &\coloneqq 0 \end{aligned}\right\} \quad pre([\,], u) \coloneqq (u = 0)$$

$$\left.\begin{aligned} \texttt{length}^\circ(x :: xs) &\coloneqq \underline{\texttt{length}^\circ(xs, u)} + 1 \\ \texttt{length}(x :: xs) &\coloneqq \underline{\texttt{length}(xs)} + 1 \end{aligned}\right\} \quad pre(x :: xs, u) \coloneqq \underline{pre(xs, u)}$$

Note how in the comparison of base cases, equality between results is enforced, whereas the joint recursive underlined calls translate into a recursive call in pre.

This predicate can finally be **replaced** by expression $u = 0$ over its static argument u leading to conditional equation (7). ■

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 $\overline{p}_i = p_1, \ldots, p_n$,[2] 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 fg (e.g. $\texttt{length}_{+\!+}$ used in the previous section).

[2] We denote vectors (of expressions, ...) by overbars, e.g., $\overline{e} = e_1, \ldots, e_n$ for some n.

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:

$$\textbf{exprs}\ e, e' \coloneqq x \mid c(\overline{e}) \mid f(\overline{e}) \quad \textbf{patterns}\ p, q \coloneqq x \mid c(\overline{p}) \quad \textbf{values}\ v \coloneqq c(\overline{v})$$

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 work.

Baseline: Enumerative Synthesis. Given a set F of typed functions/predicates $f\colon t_1, \ldots, t_n \to t \in F$ 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, wheres in the recursive case, for each function f

[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`.

from F we enumerate possible arguments of smaller depth.

$$\begin{aligned}\Sigma_0(\overline{x},t) &= \{x_i \in \overline{x} \mid x_i\colon t\} \\ \Sigma_d(\overline{x},t) &= \{\ f(e_1,\ldots,e_n) \mid f\colon t_1,\ldots,t_n \to t \in F \text{ and} \\ &\qquad\qquad e_i \in \Sigma_{d_i}(\overline{x},t_i) \text{ for } d_i < d, i = 1,\ldots,n\ \}\end{aligned} \tag{10}$$

It is often good enough to limit the number of occurrences o of each variable to $o = 2$ or $o = 3$, which cuts down the search space significantly.

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

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 the search space:

$$\textbf{theory exploration} \qquad \Gamma \vdash \varphi^? \qquad \text{where } \varphi^? \in \Sigma_d(\overline{x}, \texttt{bool}) \tag{11}$$

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 (section 2) to measure its effectiveness in relation to the search space.

In the next three sections, we present our novel scalable algorithms used for theory exploration. Section 7 then brings them together aiming to deepen the theory exploration. Although our key *contribution is in the combination* of the techniques, each of the ingredients benefits from novel optimizations and can be used for the scalable theory exploration alone.

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.

Definition 1 (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 2 (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 [40, Def. 5.9]. We write $p \perp e$ when there can be no such unifier, i.e., the pattern match is "refuted".*

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

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

Input: Γ, the set of definitions and known lemmas
Input: definition of f as $\{\ f(\overline{p}_i^f, q_i^f) \coloneqq e_i^f \ \texttt{if}\ \varphi_i^f\ \}_{i=1,\dots,m} \subseteq \Gamma$
Input: definition of g as $\{\ g(\overline{p}_j^g) \coloneqq e_j^g \ \texttt{if}\ \varphi_j^g\ \}_{j=1,\dots,n} \subseteq \Gamma$
Output: Δ with the definition of fg and lemma $f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y})$

1 $\Delta \leftarrow \{\ f(\overline{x}, g(\overline{y})) = fg(\overline{x}, \overline{y})\ \}$
2 **for** $j \leftarrow 1, \dots, n$ 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 $\quad\quad \Delta \leftarrow \Delta \cup \{\ fg(\overline{x}, \overline{p}_j^g) \coloneqq e'\ \texttt{if}\ \varphi_j^g\ \}$
6 $\quad$ **else**
7 $\quad\quad$ **for** $i \leftarrow 1, \dots, m$ cases in the definition of f **do** unfold f
8 $\quad\quad\quad$ **assert** $\mathit{free}(\overline{p}_i^f) \cap \mathit{free}(\overline{p}_j^g) = \varnothing$ (rename if needed)
9 $\quad\quad\quad$ **if** $\Gamma \vdash e_j^g \rightsquigarrow e'$ *and* $\exists\ \sigma.\ q_i^f \overset{\sigma}{\equiv} e'$ *and* $\Gamma_{fg} \vdash \sigma(e_i^f) \rightsquigarrow e''$
10 $\quad\quad\quad\quad$ *so that* e'' *is in fused form wrt.* f *and* g **then** fold fg?
11 $\quad\quad\quad\quad$ **let** $\overline{p} \leftarrow \sigma(\overline{p}_i^f, \overline{p}_j^g)$ and $\varphi \leftarrow \sigma(\varphi_i^f \wedge \varphi_j^g)$
12 $\quad\quad\quad\quad \Delta \leftarrow \Delta \cup \{\ fg(\overline{p}) \coloneqq e''\ \texttt{if}\ \varphi\ \}$
13 $\quad\quad\quad$ **else if** $q_i^f \perp e'$ **then**
14 $\quad\quad\quad\quad$ **continue** (f's ith case never matches g's j-th case's result)
15 $\quad\quad\quad$ **else**
16 $\quad\quad\quad\quad \Delta \leftarrow \varnothing$ and **fail** (fusion impossible resp. currently blocked)

The fusion algorithm 1 is realized as an unfold/fold transformation [4]. 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. definition 2) between g's result and f's pattern, and if so, adds a corresponding defining case for fg.

The main concern is to achieve that the expression e'' which is ultimately used in the definition of fg (line 11) is in the fused form wrt. f and g, for which 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 example 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

alltogether. 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. appendix A.1). In the evaluation, this premise is always satisfied.

Example 4. We continue example 1 to algorithmically fuse $f = \texttt{length}$ and $g = \texttt{++}$ into $fg = \texttt{length}_{\texttt{++}}$ so that $\texttt{length}_{\texttt{++}}(xs, ys) = \texttt{length}(xs \texttt{++} ys)$. More complex examples that make use of all features of algorithm 1 are deferred to appendix C.

Unfolding $\texttt{++}$, we have one base case ($j = 1$) and one recursive case ($j = 2$). For the former, e_1^g is ys so that $f(e_1^g)$ is $\texttt{length}(ys)$ and already in fused form (line 4), which produces the base case of $\texttt{length}_{\texttt{++}}$ in example 1. If we were to unfold `length` as well in this situation, we would instead get *two* cases, as e' being ys unifies with both patterns `[]` and $x :: xs$ of `length` in line 9. Apart from destroying the correspondence between $\texttt{length}_{\texttt{++}}$ and `length` it turns that argument into a non-accumulator and thus prevents further progress.

In the recursive case $e_2^g = x :: (xs \texttt{++} ys)$ and $\texttt{length}(e_2^g) = \texttt{length}(xs \texttt{++} ys) + 1$ by definition. Here, we make use of the fold rule (line 4), where $xs \prec_{\texttt{++}} (x :: xs)$, which produces $\texttt{length}_{\texttt{++}}(xs, ys) + 1$ in 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 (definition 3). Expression $e^?$ compensates for the absense 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 (definition 4).

To make the recursive calls in the body of a function explicit, we denote each i-th case $f(\overline{p}_i, u) \coloneqq e_i \texttt{ if } \varphi_i$ 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 $a_i^j(u)$.

$$e_i = b_i\big(\, f\big(e_i^1, a_i^1(u)\big), \ldots, f\big(e_i^k, a_i^k(u)\big) \,\big) \tag{12}$$

Definition 3 (Accumulator). *A parameter is an accumulator of f if it is 1) matched by a variable u in each pattern, 2) it specifies the values a_i^k 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 4 (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, u of $\texttt{length}^\circ$ in section 2 is static.

Algorithm 2 shows our algorithm for accumulator removal. It generates f' case-by-case from the definition of f by matching its recursive structure, but

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

Input: Γ, the 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: Δ with a sketch of the definition of f' and $f(\overline{x}, u) = e^?(f'(\overline{x}), \overline{x}', u)$

1 $\Delta \leftarrow \{\ f(\overline{x}, u) = e^?(f'(\overline{x}), \overline{x}', u)\ \}$
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. (12))*
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 $\Delta \leftarrow \Delta \cup \{\ f'(p_i) \coloneqq b_i'\big(\ f(e_i^1), \dots, f(e_i^k)\ \big) \texttt{ if } \varphi_i\ \}$

by allowing different body expressions b_i'. Note patterns p_i and guards φ are kept to preserve the recursive traversal, so that arguments matching the i-th case of f match exactly iff they same case of f'. For that reason, the removed accumulator may not occur in guards in the first place (cf. definition 3). 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 [15], but it is formulated in a more straight-forward way.

The algorithm is presented nondeterminstically 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 example 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 example 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 (10)). 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.

Example 5. Accumulator removal for $f = \texttt{length}^\circ$ yields lemma $\texttt{length}^\circ(xs, u) = \texttt{length}(xs) + u$. Algorithm 2 fills in the sketch for f' with $f'(\texttt{[]}) \coloneqq b'_1$ and $f'(x :: xs) \coloneqq b'_2(x, xs, f'(xs))$. The base case is solved by $b'_1 = c$ for $c = 0$ as a known neutral element of $\oplus = +$, so that $e^?(y, u) = y + u$. In the recursive case for that $e^?$, taking $b'_2(y, x, xs,) = b_2(y, x, xs) = y + 1$—where y represents the result of the recursive call (cf. line 10). The condition $(y + 1) + u = (y + u) + 1$ in line 13 is validated by properties of $+$. ■

Example 6. The base-case condition $\texttt{length}(ys) = e^?(b'_1, ys)$ arising from removing the accumulator of $\texttt{length}_{+\!+}$ in example 2 is more complicated than in example 5, the solution is $b'_1 = 0$ and $e^?(n, ys) = n + \texttt{length}(ys)$ for static ys. ■

6 Calculating Synthetic Preconditions

As illustrated in section 2, conditional lemmas are generated with from a case-by-case analysis of comparing the definitions of two functions, from which a synthetic precondition predicate is extracted so that:

$$pre(x, \overline{y}, \overline{z}) \implies f(x, \overline{y}) = g(x, \overline{z}) \tag{13}$$

Here, xs is a subset of the arguments of focus, which must be shared by f and g (i.e., compatible types). Conceptually, inferring preconditions like this can be seen as a form of higher-order unification in which the unifier *pre* is not just a substitution but more complex itself.

Algorithm 3 pairs each case of f with each case of g assuming that these are defined over mutually disjoint variables. The match possible if there is an overlap of the respective patterns of the shared arguments (line 5), i.e., there are values that are accepted by both cases. If not, the corresponding case in the precondition is not feasible and we can ignore it. Otherwise, similar to fusion, we use the substitution σ to narrow down the situation and to define a corresponding case of *pre* in line 8. This relies on a helper function to compute when the respective bodies e_i^f and e_j^g become equal, wherein a paired recursive call of f and g is turned to a recursive call to *pre*.

Definition 5 (Recursive Intersection of Expressions). *For two expressions e and e', their intersection $e \overset{pre}{\sqcap} e'$ with respect to pre computes a condition under which e and e' yield the same result:*

$$c(\overline{e}) \overset{pre}{\sqcap} d(\overline{e}') = \texttt{false} \qquad \text{for constructors } c \neq d \tag{14}$$

$$h(\overline{e}) \overset{pre}{\sqcap} h(\overline{e}') = \overline{e} \overset{pre}{\sqcap} \overline{e}' \qquad \text{for any } h \notin \{f, g\} \tag{15}$$

$$f(\overline{a}, \overline{e}) \overset{pre}{\sqcap} g(\overline{a}', \overline{e}') = \overline{a} = \overline{a}' \wedge pre(\overline{a}, \overline{e}, \overline{e}') \qquad \text{compatible recursive calls} \tag{16}$$

$$e \overset{pre}{\sqcap} e' = e = e' \qquad \text{if } f \notin e \text{ and } g \notin e' \tag{17}$$

$$e \overset{pre}{\sqcap} e' = \texttt{false} \qquad \text{otherwise} \tag{18}$$

Algorithm 3: Algorithm $\textsc{Match}(\Gamma, f, g)$. Arguments $\overline{x}$ merged between f and g are assumed to be the first ones without loss of generality.

Input: Γ, the set of definitions and known lemmas
Input: definition of f as $\{\ f(\overline{p}_i^f, \overline{q}_i^f) \coloneqq e_i^f \ \texttt{if}\ \varphi_i^f\ \}_{i=1,\dots,m} \subseteq \Gamma$
Input: definition of g as $\{\ g(\overline{p}_j^g, \overline{q}_j^g) \coloneqq e_j^g \ \texttt{if}\ \varphi_j^g\ \}_{j=1,\dots,n} \subseteq \Gamma$
Output: Δ with *pre* and lemma $pre(\overline{x}, \overline{y}, \overline{z}) \implies f(\overline{x}, \overline{y}) = g(\overline{x}, \overline{z})$

$\Delta \leftarrow \{\ pre(\overline{x}, \overline{y}, \overline{z}) \implies f(\overline{x}, \overline{y}) = g(\overline{x}, \overline{z})\ \}$
for $i \leftarrow 1, \dots, m$ (cases in the definition of f) **do**
  for $j \leftarrow 1, \dots, n$ (cases in the definition of g) **do**
    assert $free(\overline{p}_i^f) \cap free(\overline{p}_j^g) = \varnothing$ (rename if needed)
    if $\exists\ \sigma$ *with* $\overline{p}_i^f \overset{\sigma}{\equiv} \overline{p}_j^g$ the most general unifier of shared arguments **then**
      let $\overline{p} \leftarrow \sigma(\overline{p}_i^f)$ $\quad (= \sigma(\overline{p}_j^g)$, cf.definition 2)
      let $e \leftarrow \sigma(e_i^f) \overset{pre}{\sqcap} \sigma(e_j^g)$ and $\varphi \leftarrow \sigma(\varphi_i^f \wedge \varphi_j^g)$
      $\Delta \leftarrow \Delta \cup \{\ pre(\overline{p}, \overline{y}, \overline{z}) \coloneqq e \ \texttt{if}\ \varphi\ \}$
if Δ has no base case or no recursive case **then**
  $\Delta \leftarrow \varnothing$ and **fail**

where in (15) h *can be any function or constructor and*

$$(e_1, \dots, e_k) \overset{pre}{\sqcap} (e'_1, \dots, e'_k) \ = \ (e_1 \overset{pre}{\sqcap} e'_1) \wedge \dots \wedge (e_k \overset{pre}{\sqcap} e'_k)$$

compares the two argument lists pointwise.

For algorithm 3 to produce a sound precondition, we need to take some care. First, note that we can always be conservative, e.g., in (18) when there is a syntactic mismatch that is not necessarily semantic. Dually, we rely on our assumption that the cases of f and g cover all inputs. If we omit certain defining cases to achieve underspecification, as it can be done in some systems where definitions are given as axioms, *pre* will inherit this underspecification, and the generated lemma will hold only for *some* interpretations of *pre*. Then, generating the same *pre* from different comparisons could lead to inconsistencies.

7 Main Algorithm

The main algorithm 4 saturates a database Γ of definitions and discovered lemmas by repeatedly applying the transformation on original as well as synthetic functions. Algorithms Fuse (described in section 4) and RemoveAcc (described in section 5) return an equation of the corresponding shape as shown above if they succeed, together with the defining equations of the respective synthetic functions. Likewise, Match (described in section 6) returns conditional lemmas together with the definitions of synthetic preconditions.

The three steps (fusion, accumulator removal, conditional lemmas) may benefit from the accumulated set of lemmas Γ, as shown in example 10 for fusion. Our algorithm retries failed steps as long as new information can be gained.

Algorithm 4: Lemma synthesis by saturation using fusion, accumulator removal, and recognition of structurally similar functions.

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

$\Gamma \leftarrow \Delta$
repeat
  $\Gamma \leftarrow \Gamma \cup \textsc{Fuse}(\Gamma, f, g)$ for pairs of functions f and g
  $\Gamma \leftarrow \Gamma \cup \textsc{RemoveAcc}(\Gamma, f)$ for functions f with accumulators
  $\Gamma \leftarrow \Gamma \cup \textsc{Match}(\Gamma, f, g)$ for pairs of functions f and g
  $\Gamma \leftarrow \Gamma[f(\overline{x}) \mapsto \mathit{rhs}]$ if $f(\overline{x}) = \mathit{rhs}$ is a replacement (3)
$\Lambda \leftarrow \textsc{Extract}(\Gamma) \setminus \Delta$

Intermittently, the algorithm applies replacement lemmas (3): As soon as a function f is known to be the identity function or constant, or when it is structurally the same as another one, it is replaced in all lemmas and its definition is removed from Γ. This de-duplicates the effort and avoids vacuously fused forms (cf. section 4). Replacement is oriented to keep original functions if possible.

Identity functions can be recognized by a simple syntactic analysis of their defining cases. Similarly, constant functions $f(\overline{x}) = c(\overline{x}')$ can be recognized when all base cases are equal to $c(\overline{x}')$ for the same expression c over the static arguments of f (cf. definition 4). Recognizing structurally identical functions $f(\overline{x}) = h(\pi(\overline{x}))$ is more difficult [42] because one has to come up with the right permutation of arguments π and with a bijection that pairs each defining case of f with one of h. However, instead of insisting on a perfect solution, in the implementation, we heuristically canonicalize definitions for comparison based on hash functions and subsequently test for α-equivalence, which works fine.

The final step of the algorithm is to extract useful lemmas from Γ, using

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

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). *All three transformations* Fuse, RemoveAcc, *and* Match *produce valid lemmas, and new synthetic functions satisfy all assumptions of section 3 (the proofs are in appendix A).* □

As a consequence, algorithm 4 maintained the following invariants

- definitions including synthetic ones in Γ satisfy the assumptions of section 3
- lemmas $\varphi \in \Gamma$ semantically follow from the definitions Δ

Theorem 1. *Lemmas Λ returned by algorithm 4 follow definitions Δ.* □

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 [10]. Alternatively, one can represent Γ as an E-graph [11,35,51], 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.

8 Evaluation

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 (cf. appendix B). A full theory gets from 8 to 18 functions, from which our baseline enumerator generates ∼1.5M candidates in total. Interestingly, Z3 (even when equipped with external induction) can prove that only ∼0.01% of them are lemmas. A naive enumeration-based theory explorer thus spends most of its time (more than 13 hours, in total, cf. table 1 in appendix B) when dealing with false candidates. An individual benchmark takes from 4 to 7 functions which give rise to ∼265K candidates, in total. The success rate of the enumerator in this setting is substantially higher here than for the full theories: ∼1.7% of candidates were proven to be lemmas, and it takes slightly over one hour. We treat this experiment as "auxiliary" because it does not directly evaluate any of our contributions, and the results are used only to judge the difficulty of benchmarks.

We compare four approaches: Struct: LemmaCalc based on structural transformations, but without conditional equalities (sections 4 and 5, Cond: LemmaCalc including synthetic preconditions (sections 4 to 6) Enum: Our own enumerative generator, based on (11) in section 3, which solves for $e^?$ in equational lemmas $f(\_, g(\_)) = e^?$, TheSy [43], a state-of-the-art enumerative lemma generator. LemmaCalc runs multiple rounds of lemma discovery (here three) so that proofs that had failed earlier can benefit from lemmas discovered later (examples 8 and 10 in appendix C). TheSy also makes use of lemmas discovered so far, and in contrast to our baseline can discover conditional lemmas.

The baseline enumerator is included to measure the coverage that could in principle be achieved by the two structural methods, fusion and removal of accumulators. The it is run over a search space up to depth $d = 3$ and maximal variable occurrence $o = 2$ in in (11), which is sufficient to cover all lemmas found by LemmaCalc, and by experience, more deeply nested lemmas tend to be consequences of simpler ones here. To keep evaluation tractable despite the large search space, we use a simple counterexample check that evaluates small ground instances of lemmas. This check is very effective to falsify a large fraction of lemma candidates (cf. table 1). We use Z3 4.12.2 with a timeout of 500ms to further rule out all lemmas that can be decided without induction as either false

or as trivially true. Only candidates that need induction will be reported: the baseline enumerator tries induction on all variables in scope before calling Z3. Lemmas found in this way are then re-used for future checks, revisiting unknown lemmas in multiple rounds helps a little, too (cf. caption of table 1). THESY is run with default settings on a translation of theories into its input format that is based on rewrite rules. From each defining equation $f(\varphi) \coloneqq e$ `if` $\overline{p}$, we generate a rewrite rule $f(\overline{p}) \rightsquigarrow e$ `if` φ in its native input format.

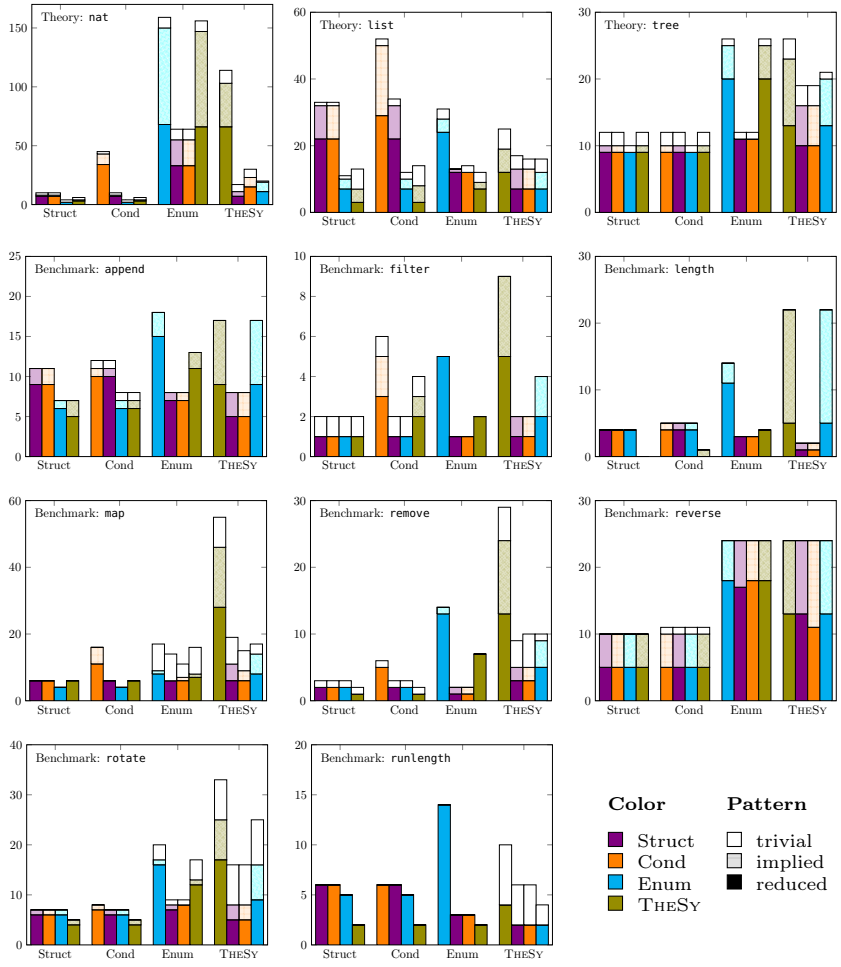

Fig. 1. Experiments for the full theories (top row) and individual benchmarks (rows below). The first bar in each group represents the number of lemmas found by the approach listed on the x-axis below, the subsequent three bars represent the proportion of these lemmas that can be confirmed by the other approaches. Bars are partitioned into trivial lemmas (white), implied lemmas (lightly shaded), and those in the reduced set (solid color). Four additional higher-order lemmas mentioning `map(succ, _)` are omitted from THESY's result in `list` because they were not supported by our toolchain.

What is the relative explanatory strength of the sets of lemmas generated? For each set of lemmas generated by a method on a benchmark, we measure how many of those are implied ("confirmed") by the lemmas found by each other method. Lemmas are moreover partitioned into three categories, *trivial* are those implied by the original theory (i.e., those do not need induction, *reduced* are those that remain when one incrementally removes lemmas that are implied by the rest, and *implied* are the nontrivial ones that were removed. We report the confirmation rate by other methods for each individually. We use Z3 with a timeout of 500ms but due to incompleteness of the proof oracle the results should not be taken being precise but rather be interpreted qualitatively.

This comparison is shown in fig. 1. As an example, on benchmark `append`, the structural method finds 11 lemmas, of which 2 are implied by the other 9, and there are no trivial lemmas. The nontrivial resp. reduced sets of lemmas of the conditional method can fully confirm these, and enumeration and TheSy cover roughly 60% of these. Overall, the results differ widely across the benchmarks. Enumeration-based methods tend to outperform LemmaCalc, which is expected as they cover a much larger space, but calculational techniques can cover a significant proportion of that space and sometimes even generate lemmas not found by the other approaches. Enumeration generates some amount of trivial and redundant (implied) lemmas (white/lightly shaded parts). This is true for LemmaCalc, too, when there are similar functions in the input theory, e.g., possibilities for fusion and accumulator removal discovered for `++` (append) will typically *also* be discovered for `snoc` (adding an element to the end of a list).

What is the impact of the size of the search space? Generally, enumeration takes longer when more functions are present, see table 1, but it also strongly depends on how many possible combinations there are. On the full theories, our baseline enumerator takes over 6 hours. 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`. Overall, the numbers confirm our intuition, that enumeration of large search spaces is expensive. Besides the time it takes, it is also not clear when to stop exploration, because lemmas are often discovered sporadically after long periods of unproductive search.

8.1 Strengths and Weaknesses, Respectively

count-append trips up prover in enumerate because add comm maybe

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.

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. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM (JACM) **24**(1), 44–67 (1977)
5. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The ACL2 Sedan Theorem Proving System. In: TACAS. pp. 291–295. Springer (2011)
6. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 277–293. Springer (2012)
7. 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)
8. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Tip: tons of inductive problems. In: International Conference on Intelligent Computer Mathematics. pp. 333–337. Springer (2015)
9. 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)
10. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: Formal models and semantics, pp. 243–320. Elsevier (1990)
11. 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`
12. Fedyukovich, G., Ernst, G.: Bridging Arrays and ADTs in Recursive Proofs. In: TACAS. LNCS, Springer (2021)
13. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling Invariants from Frequency Distributions. In: FMCAD. pp. 100–107. IEEE (2017)
14. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 255–263. IEEE (2020)
15. Giesl, J.: Context-moving transformations for function verification. In: LOPSTR. pp. 293–312. Springer (1999)
16. Giesl, J., Kühnemann, A., Voigtländer, J.: Deaccumulation techniques for improving provability. The Journal of Logic and Algebraic Programming **71**(2), 79–113 (2007)
17. 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)

18. 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
19. Grechanik, S.: Inductive prover based on equality saturation for a lazy functional language. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. pp. 127–141. Springer (2014)
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. vol. 6355, pp. 348–370. Springer (2010)
32. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based Verification for Rust Programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021)
33. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: FPCA. vol. 91, pp. 124–144 (1991)
34. Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: PLDI. pp. 1–15 (2020)
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. Ohori, A., Sasano, I.: Lightweight fusion by fixed point promotion. ACM SIGPLAN Notices **42**(1), 143–154 (2007)

37. Pham, T., Gacek, A., Whalen, M.W.: Reasoning about algebraic data types with abstractions. J. Autom. Reason. **57**(4), 281–318 (2016)
38. 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. vol. 11562, pp. 74–83. Springer (2019)
39. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: VMCAI. vol. 8931, pp. 80–98. Springer (2015)
40. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM (JACM) **12**(1), 23–41 (1965)
41. de Roever, W.P., Engelhardt, K.: Data refinement: Model-oriented proof methods and their comparison. Cambridge University Press (1998)
42. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M.: Nominal unification of higher order expressions with recursive let. In: Logic-Based Program Synthesis and Transformation: 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6–8, 2016, Revised Selected Papers 26. pp. 328–344. Springer (2017)
43. 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)
44. 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)
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. 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. 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: VMCAI. Springer (2023)

A Soundness Proofs

We are working with typed functions $f\colon t_1, \ldots, t_n \to t$, 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 6 (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 $f(\overline{p}_i) \coloneqq e_i\big(\, f(\overline{e}) \,\big)$ 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 7. *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 8. *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 9 (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 $1 \le i < j < m$.*

Definition 10 (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 definition 10 we can prove any property $P(\overline{z})$ over $\overline{z}\colon \overline{t}$ by induction:*[4]

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

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

A.1 Properties of Fuse (Algorithm 1 in Section 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*

$$
\begin{aligned}
[\![\overline{p}_i^f \,\mathtt{if}\, \varphi_i^f]\!] \cap [\![\overline{p}_{i'}^f \,\mathtt{if}\, \varphi_{i'}^f]\!] = \varnothing \qquad & \overset{\varnothing \times A = \varnothing}{\Longrightarrow} \\
([\![\overline{p}_i^f \,\mathtt{if}\, \varphi_i^f]\!] \cap [\![\overline{p}_{i'}^f \,\mathtt{if}\, \varphi_{i'}^f]\!]) \times ([\![\overline{p}_j^g \,\mathtt{if}\, \varphi_{j'}^g]\!] \cap [\![\overline{p}_j^g \,\mathtt{if}\, \varphi_{j'}^g]\!]) = \varnothing \qquad & \overset{\text{lemma 2}}{\Longleftrightarrow} \\
([\![\overline{p}_i^f \,\mathtt{if}\, \varphi_i^f]\!] \times [\![\overline{p}_j^g \,\mathtt{if}\, \varphi_{j'}^g]\!]) \cap ([\![\overline{p}_{i'}^f \,\mathtt{if}\, \varphi_{i'}^f]\!] \times [\![\overline{p}_j^g \,\mathtt{if}\, \varphi_{j'}^g]\!]) = \varnothing \qquad & \overset{\text{lemma 4}}{\Longleftrightarrow} \\
[\![\overline{p}_i^f, \overline{p}_j^g \;\mathtt{if}\; \varphi_i^f \wedge \varphi_j^g]\!] \cap [\![\overline{p}_{i'}^f, \overline{p}_{j'}^g \;\mathtt{if}\; \varphi_{i'}^f \wedge \varphi_{j'}^g]\!] = \varnothing \qquad & \overset{\text{lemmas 3 and 5}}{\Longrightarrow} \\
[\![\sigma(\overline{p}_i^f, \overline{p}_j^g) \;\mathtt{if}\; \sigma(\varphi_i^f \wedge \varphi_j^g)]\!] \cap [\![\sigma(\overline{p}_{i'}^f, \overline{p}_{j'}^g) \;\mathtt{if}\; \sigma(\varphi_{i'}^f \wedge \varphi_{j'}^g)]\!] = \varnothing &
\end{aligned}
$$

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 wrt. 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 definition 10 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 algorithm 1.

By pattern completeness of g, there is a case j of g with $\overline{w} \in [\![\overline{p}_j^g \;\mathtt{if}\; \varphi_j^g]\!]$, and by definition 8 there is a substitution τ^g over $\mathit{free}(\overline{p}_j^g)$ with

$$\tau^g(\overline{p}_j^g) = \overline{w} \quad \text{and} \quad \tau^g(\varphi_j^g) \text{ holds} \tag{19}$$

If case j of g can be processed by line 4, we have $[\![\overline{x}, \overline{p}_j^g \;\mathtt{if}\; \varphi_j^g]\!] = V_{\overline{t}^f} \times [\![\overline{p}_j^g \;\mathtt{if}\; \varphi_j^g]\!]$. 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., $\overline{v}, \tau^g(e_j^g) \in [\![\overline{p}_i^f, q^f \;\mathtt{if}\; \varphi_i]\!]$ and by definition 8 there is a substitution τ^f over $\mathit{free}(\overline{p}_j^f, q^f)$ with

$$\tau^f(\overline{p}_i^f, q_i^f) = \overline{v}, \tau^g(e_j^g) \quad \text{and} \quad \tau^f(\varphi_i^f) \text{ holds} \tag{20}$$

Note, τ^f and τ^g are over disjoint variables by the condition in line 8 of algorithm 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 (19) and (20) 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 $e' = e'^g_j$ (by soundness of rewriting) and the most general unifier σ. Definition 2 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 definition 8 is given as τ':

$$\tau'(\overline{p}) = \tau'(\sigma(\overline{p}^f_i, \overline{p}^g_j)) = \tau'(\sigma(\overline{p}^f_i), \tau'(\sigma(\overline{p}^g_j)) = \tau'_f(\sigma(\overline{p}^f_i), \tau'_g(\sigma(\overline{p}^g_j)) = \overline{v}, \overline{w}$$

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^g_j, for line 4 we have

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

For line 9, assuming $\sigma(\varphi^f_i \wedge \varphi^g_j)$ we have

$$fg(\sigma(\overline{p}^f_i, \overline{p}^g_j)) \overset{\text{def. } fg}{=} e' \overset{\Gamma_{fg} \text{ valid}}{=} \sigma(e^f_i) \overset{\text{def. } f}{=} f(\sigma(\overline{p}^f_i), \sigma(q^f_i))$$
$$\overset{\text{def. } 2}{=} f(\sigma(\overline{p}^f_i), e') \overset{\Gamma \text{ valid}}{=} f(\sigma(\overline{p}^f_i), e^g_j) \overset{\text{def. } g}{=} f(\sigma(\overline{p}^f_i), g(\sigma(\overline{p}^g_j)))$$

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. 2 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. □

A.2 Properties of RemoveAcc (Algorithm 2 in Section 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*

$$\forall\ \overline{x}.\ \ \big(\forall\ u.\ \overline{e}, a^j_i(u) \prec_{f'} \overline{p}_i, u\big) \implies \overline{e} \prec_{f'} \overline{p}_i$$

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. □

A.3 Properties of Match (Algorithm 3 in Section 6)

Lemma 15 (Termination). *pre terminates with* $\overline{x}', \overline{y}', \overline{z}' \prec_{pre} \overline{x}, \overline{y}, \overline{z}$ *defined to hold if* $\overline{x}', \overline{y}' \prec_f \overline{x}, \overline{y}$ and $\overline{x}', \overline{z}' \prec_g \overline{x}, \overline{z}$. □

Lemma 16 (Pattern Disjointness and Disjointness). *The proofs go analogous to that for Fuse in dealing with the substitutions. We omit the details.* □

Lemma 17. *If* $P \equiv \big(pre(\overline{x}', \overline{y}', \overline{z}') \implies f(\overline{x}', \overline{y}') = g(\overline{x}', \overline{z}')\big)$ *holds for all* $f(\overline{x}', \overline{y}')$ *that occur in* e *and all* $g(\overline{x}', \overline{z}')$ *that occur in* e'*, then* $e \overset{pre}{\sqcap} e'$ *(definition 5) implies* $e = e'$.

Proof. By induction on how $e \overset{pre}{\sqcap} e'$ is computed (i.e., size of e, e'). Cases (14) and (18), hold vacuously, as the premise is `false`. Case (15) follows from the inductive hypothesis for the respective arguments of function h. Case (17) simply enforces the desired conclusion $e = e'$. Finally, case (16) refers to assumption P of this lemma that all pairs of recursive calls are already matched by *pre*. □

Lemma 18. *Conditional lemma* $pre(\overline{x}, \overline{y}, \overline{z}) \implies f(\overline{x}, \overline{y}) = g(\overline{x}, \overline{z})$ *holds.*

Proof. By induction over the computation of *pre* (lemma 6 for $\prec_{pre}$), noting that this relies on pattern completeness of *pre*, which in turn only holds because f and g have complete patterns already. We make use of lemma 17 for the construction of line 7. It's assumption P is an instance of the inductive hypothesis for the σ computed in line 5 of algorithm 3. □

B Benchmark Overview

Input files for all benchmarks are provided in Boogie-like syntax as well as in SMT-LIB as part of the supplementary material.

Table 1 shows the number of function symbols $|F|$ in each theory, and the respective search space covered by our baseline enumerator, which is implements (11) from section 3 (further detailed below). The table shows some statistics on the status of these lemmas, where column "lemma" are those true formulas that need inductive proofs. Due to incompleteness of the proof oracle, table 1 gives a lower estimate of the candidates that are valid in that search space.

Functions present in the respective benchmarks:

- `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 logical negation. `add`, `sub`, `mul`, `leq`, `lt` are structurally recursive definitions over natural numbers for $+$, $-$, $*$, $\leq$, and $<$. `filter` keeps elements that satisfy a given predicate, `countif` counts them, and `all`/`ex` test if all/some element satisfies a given predicate. `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.[5] A critical lemma connects `sum` over the decoded sequence to `sumruns` that works on the coded one.

C Detailed Examples

This section shows some examples in full detail. We furthermore include some situations where our approach fails and discuss how this is linked to inherent limitations resp. issues with the current implementation.

Example 7. Associativity of append is discovered[6] by fusing $(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$ into a synthetic function $\mathbin{+\!\!+\!\!+}(xs, ys, zs)$ and then by deaccumulating its last argument. The correct choices are $b'_1 = c =$ `[]` as *right*-neutral element of $\oplus = \mathbin{+\!\!+}$ and canoncially $b'_2 = b_2 =$ `_::_` as the original function body of the outer $\_ \mathbin{+\!\!+} zs$. ■

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

[6] Both examples in this section are presented with minimal detail only due to lack of space. The reader is invited to follow along on a piece of paper—by our experience this is crucial to get some intutition into how the two transformations work together.

Table 1. Statistics on benchmark theories used in the comparison, where $|F|$ is the number of functions. The number of candidates is $\Sigma_{f(\overline{x},g(\overline{y}))}|S_3^2(\overline{x},\overline{y},\texttt{bool})|$, i.e., potential right-hand sides to equations (11) of depth $d = 3$ and max $o = 2$ occurrences of each variable. false: falsified random testing or by Z3, true: proved by Z3, lemma: proved by induction+Z3, and candidates with unknown status. The timeout for Z3 was 50ms per query. Time is shown in hours:minutes:seconds for the first round of enumeration. Marked † results contain lemmas found in later rounds (round 2: map (1), runlength (4), nat (14); round 3 tree (1)). In comparison to the runtimes, LemmaCalc with conditional matching takes 14s `list` as the largest benchmark, all other results are produced within 1s–5s. 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, in most cases except for `nat`, TheSy stopped reporting lemmas long before that.

		baseline enumerator statistics						TheSy	
benchmark	$\|F\|$	candidates	false	true	lemma	unknown	time	last	killed
`nat`	8	1 131 799	1 129 504	309	159†	1 827	09:53	26:38:14	>26h
`list`	18	320 978	203 993	384	31	116 569	6:21:16	10:55:14	>21h
`tree`	11	123 488	107 569	118	26†	15 776	6:31:37	16:47	
`append`	5	15 295	12 584	128	18	2 564	10:13	04:32	
`filter`	6	398	75	2	5	319	00:39	00:02	
`length`	5	7 066	6 495	556	14	1	00:22	00:00	
`map`	6	17 721	14 494	31	17†	3 179	10:08	37:33	>11h
`remove`	7	32 916	24 059	121	14	8 722	54:16	13:01	>11h
`reverse`	4	127 926	127 476	425	24	1	07:45	00:02	
`rotate`	6	12 784	12 597	123	21	43	00:34	6:54:22	>11h
`runlength`	6	68 311	67 499	221	27†	564	06:39	00:40	>11h

Example 8 (Reversing Lists). We show how deaccumulation is key to discovering the classic lemma $\texttt{reverse}(\texttt{reverse}(xs)) = xs$ in our approach (the variant using `snoc` works too).

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

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

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

This function can be deaccumulated, 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 $f' \equiv \texttt{reverse}$. This lemma unblocks fusion of $\texttt{reverse}(\texttt{reverse}(\_))$ via the shortcut in line 4 of algorithm 1. The fused function `reverse_reverse` can subsequently be recognized as the identity function after some simplifications. ■

Example 9 (Reordering of Arguments). An example where matching functions modulo reordering of parameters is important is when fusing `map` and `take` both ways. For permutation $\pi(1,2,3) = (2,1,3)$ we have $\texttt{map\_take}(f, n, xs) \overset{\pi}{\equiv} \texttt{take\_map}(n, f, xs)$.

Example 10 (Binary Trees). The data type for binary trees with elements of type `Elem` is

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

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.

$$\texttt{size}(\texttt{leaf}) \coloneqq 0 \qquad \texttt{size}(\texttt{node}(l, x, r)) \coloneqq \texttt{size}(l) + \texttt{size}(r) + 1 \tag{21}$$

$$\texttt{elems}(\texttt{leaf}) \coloneqq [\,] \qquad \texttt{elems}(\texttt{node}(l, x, r)) \coloneqq x :: (\texttt{elems}(l) +\!\!+ \texttt{elems}(r)) \tag{22}$$

Goal is to fuse $\texttt{length\_elems}$ 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 (22) corresponding to line 2 in algorithm 1. For the base case, $e_0^g = [\,]$ and $\Gamma \vdash \texttt{length}([\,]) \rightsquigarrow 0$ in line 4 of algorithm 1 by the definition of `length` that is part of Γ, so that $\texttt{length\_elems}(\texttt{leaf}) \coloneqq 0$.

In the recursive case, $e_1^g = e' = x :: (\texttt{elems}(l) +\!\!+ \texttt{elems}(r))$. Pattern match of the base case of `length` is refuted by $[\,] \bot e'$ (line 10 of algorithm 1). For pattern $y :: ys$ of the recursive case of `length` we get a unifier σ with $\sigma(y) = x$ and $\sigma(ys) = \texttt{elems}(l) +\!\!+ \texttt{elems}(r)$. We then look at $\sigma(\texttt{length}(ys) + 1) = \texttt{length}(\texttt{elems}(l) +\!\!+ \texttt{elems}(r)) + 1$. 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 (6) to unblock the situation which is done by the second use of rewriting in line 9 as $\texttt{length}(\texttt{elems}(l) +\!\!+ \texttt{elems}(r)) + 1 \rightsquigarrow e''$ 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 11 (Conditional Lemmas for `filter`*).* Functions like `remove` and `filter` that contain guard expressions lead to interesting precondition with respect to eqs. (4) and (5). Consider the following definition

$$\texttt{filter}(p, [\,]) \coloneqq [\,] \tag{23}$$

$$\texttt{filter}(p, y :: ys) \coloneqq y :: \texttt{filter}(p, ys) \qquad \texttt{if } p[x] \tag{24}$$

$$\texttt{filter}(p, y :: ys) \coloneqq \texttt{filter}(x, ys) \qquad \texttt{if } \neg p[x] \tag{25}$$

We can enfore a precondition $pre_{\texttt{id}}(x, xs)$ so that $\texttt{filter}(p, xs) = xs$ becomes the identity function. Suppose we have

$$\texttt{id}([\,]) \coloneqq [\,] \qquad\qquad \texttt{id}(z :: zs) \coloneqq z :: \texttt{id}(zs)$$

Then algorithm 3, executed for sharing the second argument of `filter` with the argument of `id`, produces the following pairings of cases:

$$
\begin{aligned}
pre_{\texttt{id}}(p, \texttt{[]}) &\coloneqq \texttt{[]} \sqcap \texttt{[]} = \texttt{true} \\
pre_{\texttt{id}}(x, y :: ys) &\coloneqq y :: \texttt{filter}(x, ys) \sqcap y :: \texttt{id}(ys) = (y = y) \wedge pre_{\texttt{id}}(p, ys) && \texttt{if } p[x] \\
pre_{\texttt{id}}(p, y :: ys) &\coloneqq \texttt{filter}(p, ys) \sqcap y :: \texttt{id}(ys) = \texttt{false} && \texttt{if } \neg p[x]
\end{aligned}
$$

where in both recursive cases, $\sigma = \{y \mapsto z, ys \mapsto zs\}$ is the subsitution that unifies the constructor pattern. The last case is generated by (18) ("we don't know"), whereas the middle case is generated by (15) to split up the common top-level symbol (constructor ::) and subsequently by (16) which introduces the recursive call.

When we simplify this definition, we observe that $pre_{\texttt{id}}(x, xs) \equiv \texttt{all}(p, xs)$ and as a consequence, we have this lemma:

$$\texttt{all}(p, xs) \implies \texttt{filter}(p, xs) = xs$$

where function `all` tests whether all element in the list satisfies that predicate. Conversely, if we instead synthesize a precondition $pre_{\texttt{[]}} \equiv \texttt{not\_ex}(p, xs)$, that identical to fusing `ex` into `not`,

$$\neg\texttt{ex}(p, xs) \implies \texttt{filter}(x, xs) = \texttt{[]}$$

Example 12. Conditional lemmas are useful for fused functions, too. For example, consider function `take`, which retains a prefix of list xs up to length n:

$$
\begin{aligned}
\texttt{take}(0, xs) &= \texttt{[]} \\
\texttt{take}(n+1, \texttt{[]}) &= \texttt{[]} \\
\texttt{take}(n+1, y :: ys) &= x :: \texttt{take}(n, ys)
\end{aligned}
$$

Synthesizing recursive preconditions can help to deal with the different conditions of termination of this function, i.e., whether the list is long enough to have a proper prefix of a given length. Our approach can discover

$$n \leq \texttt{length}(xs) \implies \texttt{length}(\texttt{take}(n, xs)) = n$$

which it is generated with the help of intermediate fused functions for $\leq$ and `length` as well as `length` and `take`. We get a dual lemma, which not just guarantees a certain length but the actual content, too:

$$\texttt{length}(xs) \leq n \implies \texttt{take}(n, xs) = xs$$