Proving Functional Program Equivalence via Directed Lemma Synthesis

Yican Sun[1], Ruyi Ji[1], Jian Fang[1], Xuanlin Jiang[1], Mingshuai Chen[3], and Yingfei Xiong[1,2(✉)]

[1] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China
{sycpku,jiruyi910387714,xiongyf}@pku.edu.cn
{fangjian,xljiang}@stu.pku.edu.cn
[2] Zhongguancun Laboratory
[3] Zhejiang University, Hangzhou, China
m.chen@zju.edu.cn

Abstract. Proving equivalence between functional programs is a fundamental problem in program verification, which often amounts to reasoning about *algebraic data types* (ADTs) and compositions of *structural recursions*. Modern theorem provers provide *structural induction* for such reasoning, but a structural induction on the original theorem is often insufficient for many equivalence theorems. In such cases, one has to invent a set of lemmas, prove these lemmas by additional induction, and use these lemmas to prove the original theorem. There is, however, a lack of systematic understanding of what lemmas are needed for inductive proofs and how these lemmas can be synthesized automatically. This paper presents *directed lemma synthesis*, an effective approach to automating equivalence proofs by discovering critical lemmas using program synthesis techniques. We first identify two *induction-friendly* forms of propositions that give formal guarantees to the progress of the proof. We then propose two tactics that synthesize and apply lemmas, thereby transforming the proof goal into induction-friendly forms. Both tactics reduce lemma synthesis to a set of independent and typically small program synthesis problems that can be efficiently solved. Experimental results demonstrate the effectiveness of our approach: Compared to state-of-the-art equivalence checkers employing heuristic-based lemma enumeration, directed lemma synthesis saves 95.47% runtime on average and solves 38 more tasks over an extended version of the standard benchmark set.

Keywords: Program equivalence checking · Functional programs · Lemma synthesis

1 Introduction

Automatically proving the equivalence between functional programs is a fundamental problem in program verification. On the one hand, it is the basic way to certify the correctness of optimizing functional programs. On the other hand, since modern theorem provers such as Isabelle [27], Coq [1], and Lean [22] are

based on functional programming languages, many other verification problems reduce to reasoning about equivalence between functional programs.

The core of functional programming languages is built upon *algebraic data types* (ADTs). An ADT describes composite data structures by combining simpler types; it can be recursive when referring to itself in its own definition. ADTs are often processed by *structural recursions*, where recursive calls are invoked over the recursive substructures of the input value. As a result, the crux of verifying functional program equivalence is to reason about the *equivalence between composed structural recursions*, as demonstrated by the following example.

```
Inductive List = nil | cons Int List;

Let rev (l:List) =
match l with
| nil → nil
| cons h t → snoc h (rev t)
end;

Let sort (l:List) =
match l with
| nil → nil
| cons h t → ins h (sort t)
end;

Let sum (l:List) =
match y with
| nil → 0
| cons h t → h + (sum t)
end;
```

```
Let snoc (x:Int) (l:List) =
match l with
| nil → cons x nil
| cons h t → cons h (snoc x t)
end;

Let ins (x:Int) (l:List) =
match l with
| nil → cons x nil
| cons h t →
    if x ≤ h then cons x l
    else cons h (ins x t)
end;
```

Fig. 1. An algebraic data type and structurally recursive functions.

Example 1. Fig. 1 depicts a common ADT `List` with two constructors, `nil` and `cons`, and standard structurally recursive functions, `rev` that reverses a list, `sort` that applies insertion sort, and `sum` that calculates the sum of a list. Functions `snoc` and `ins` are for implementing these functions. We are interested in proving that summing a list after reverse is the equivalent of summing a list after sorting:

$$\forall\, \texttt{xs} : \texttt{List}.\quad \texttt{sum (rev xs)} = \texttt{sum (sort xs)}\,. \tag{†}$$

To prove the equivalence, it is natural to apply *structural induction*, which has been integrated into modern theorem provers. A structural induction certifies that proposition $P(x)$ holds for every instance x of some ADT by showing that $P(x)$ holds for each possible constructor of x, assuming the *induction hypothesis* that $P(x')$ holds for the substructure x' of x. For example, a structural induction for (†) requires to prove two subgoals, each corresponds to a constructor of `List`. The first subgoal is to show (†) holds when `xs = nil`. The second subgoal induces the following inductive hypothesis.

$$\texttt{sum (rev t)} = \texttt{sum (sort t)}\,. \tag{IH}$$

Proposition (†) holds for the `cons` case if: (†) is true, assuming `xs` = `cons h t` and (IH). ◁

Challenge: Lemma Finding. Nonetheless, *many theorems cannot be proved by only induction over the original theorem* [12]. Example 1 is such a case: Its proof requires induction, but induction over (†) is insufficient since we cannot apply the inductive hypothesis (IH); see the full version [34] for a formal proof. To apply (IH), we have to transform (†) until there is a subterm matching either the left-hand-side (LHS) or right-hand-side (RHS) of (IH), such that we can apply (IH) to rewrite the transformed formula. However, such a subterm can never be derived through a deductive transformation (Details in Sect. 2)

In such cases, it is necessary to invent a set of lemmas, prove these lemmas by additional induction, and use these lemmas to prove the original proposition. Accordingly, the proof process boils down to (i) *lemma finding*, and (ii) *deductive reasoning with the aid of lemmas*. Whereas decision procedures for deductive reasoning have been extensively studied [3,21,25], *there is still a lack of systematic understanding of what lemmas are needed for inductive proofs and how these lemmas can be synthesized automatically*.

Due to the lack of theoretical understanding, many existing automatic proof approaches resort to *heuristic-based lemma enumeration* [4,7,11,20,26,29–32]. These approaches typically work as follows: (i) use heuristics to rank all possible lemma candidates in a syntactic space (the heuristics are commonly based on certain machine-learning models or the textual similarity to the original proposition), (ii) enumerate the candidates by rank and (iii) try to prove each lemma candidate and certify the original proposition using the lemma. Since there is no guarantee that the lemma candidates are helpful in advancing the proof, such solvers may waste time trying useless candidates, thus leading to inefficiency. For Example 1, the enumeration-based solver HipSpec [4] produces lemma $\forall$`xs`. `rev` (`rev` `xs`) = `xs`, which provides little help to the proof.

Approach. We present *directed lemma synthesis* to avoid enumerating useless lemmas. From Example 1, we can see that the key to the inductive proof lies in the *effective application* of the inductive hypothesis. Based on this observation, we identify two syntactic forms of propositions that guarantee the effective application of the inductive hypothesis, termed *induction-friendly forms*. Next, we propose two tactics that synthesize and apply lemmas. The lemmas synthesized by our tactics take the form of an equation, with one of its sides matching a term in the original proposition, and can be used to transform the original proposition by rewriting the matched term into the other side of the lemma. Consequently, the current proof goal splits into two subgoals – one for proving the transformed proposition and the other for proving the synthesized lemma itself. Our tactics have the following properties:

- *Progress*: The new proof goals after applying our tactics eventually fall into one of the induction-friendly forms. That is, compared with existing directionless lemma enumeration, our synthesis procedure is *directed*: it eventually produces subgoals that admit effective applications of the inductive hypothesis.
- *Efficiency*: The lemma synthesis problem in our tactics can be reduced to a set of independent and typically small *program synthesis* problems, thereby allowing an off-the-shelf program synthesizer to efficiently solve the problems.

Based on the two tactics, we propose AutoProof, an automated approach to proving the equivalence between functional programs by *combining any existing decision procedure with our two tactics for directed lemma synthesis*.

For Example 1, AutoProof synthesizes the lemma

$$\forall\ \texttt{xs} : \texttt{List}.\ \ \texttt{sum (rev xs)} = \texttt{sum xs}\ ,$$

where the LHS matches the LHS of the original proposition (†). Therefore, we can use this lemma to rewrite (†) into

$$\forall\ \texttt{xs} : \texttt{List}.\ \ \texttt{sum xs} = \texttt{sum (sort xs)}\ .$$

As will be shown later, both equations above fall into the first induction-friendly form, thus ensuring the application of the inductive hypothesis.

Evaluation. We have implemented AutoProof on top of Cvc4Ind [30] – the available state-of-the-art equivalence checker with heuristic-based lemma enumeration. We conduct experiments on the program equivalence subset of an extended version of the standard benchmark in automated inductive reasoning. The results show that, compared with the original Cvc4Ind, our directed lemma synthesis saves 95.47% runtime on average and help solve 38 more tasks.

Contributions. The main contributions of this paper include the follows.

- The idea of *directed lemma synthesis*, i.e., synthesizing lemmas to transform the proof goal into desired forms.
- Two *induction-friendly forms* that guarantee the effective application of the inductive hypothesis, as well as two *tactics* that synthesize and apply lemmas to transform the proof goal into these forms. The lemma synthesis in our tactics can be reduced to a set of independent and typically small synthesis problems, ensuring the efficiency of the lemma synthesis.
- The implementation and evaluation of our approach, demonstrating the effectiveness of our approach in synthesizing lemmas to improve the state-of-the-art decision procedures.

Due to space limitations, we relegate the details to the full version [34].

2 Motivation and Approach Overview

In this section, we illustrate AutoProof over examples. For simplicity, we consider only structurally recursive functions with one parameter in this section.

A Warm-up Example. To begin with, let us first consider an equation where the direct structural induction yields an effective application of the inductive hypothesis.

$$\forall \texttt{xs} : \texttt{List}.\ \ \texttt{sum (rev xs)} = \texttt{sum xs} \qquad (\dagger_W)$$

To prove this equation, we conduct a structural induction on `xs`, the ADT argument that the structural recursion traverses, resulting in two cases $\texttt{xs} = \texttt{nil}$ and $\texttt{xs} = \texttt{cons h t}$. The first case is trivial, and in the second case, we have an inductive hypothesis over the tail list `t`.

$$\texttt{sum (rev t)} = \texttt{sum t} \qquad (\text{IH}_W)$$

We first use the equation `xs = cons h t` to rewrite the original proposition ($\dagger_W$), and obtain the following equation.

```
sum (rev (cons h t)) = sum (cons h t)
```

Here `sum` and `rev` are both structural recursions, which use pattern matching to choose different branches based on the constructor of `xs`. With `xs` replaced as `cons h t`, we can now proceed with the pattern matching and obtain the following equation.

```
sum (snoc h (rev t)) = h + (sum t)                    (1)
```

Now the equation contains a subterm `sum t` that matches the RHS of the inductive hypothesis (IH_W), which allows us to rewrite this equation with (IH_W), resulting in the following equation.

```
sum (snoc h (rev t)) = h + (sum (rev t))              (2)
```

There is a common "`rev t`" term on both sides of the equation above, and we can apply the standard generalization technique to replace it with a new fresh variable `r`, obtaining the following equation.

```
sum (snoc h r) = h + (sum r)                          (3)
```

This equation is simpler than the original one as `snoc` does not involve calls to other structurally recursive functions. By further applying induction on `r`, we can prove this equation.

We can see that the above proof contains two key steps: (i) using the inductive hypothesis to rewrite the equation, and (ii) using generalization to eliminate a common non-leaf subprogram. We call such two steps an *effective application* of the inductive hypothesis. Note that an effective application is guaranteed because the RHS of the original equation is a single structural recursion call, `sum xs`. Since a structural recursion applies itself to the substructure of the input, `sum t` is guaranteed to appear after reduction. Then, we can use the inductive hypothesis to rewrite, and the rewritten RHS contains `rev t`. Similarly, the inner-most function call, `rev xs`, is guaranteed to reduce to `rev t`. Therefore, a generalization is guaranteed.

Induction-friendly forms. In general, we identify *induction-friendly* forms, where for every equation in this form, there exists a variable such that performing induction on it yields an effective application of the inductive hypothesis for the cases involving a recursive substructure. From the discussion above, we have the simplified version of the first induction-friendly form.

(F0) *(Simplified (F1)).* One side of the equation is a single call to a structurally recursive function.

A Harder Example. Now let us consider the example equation (†) we have seen in Sect. 1. Recall this equation as follows.

```
∀ xs : List.  sum (rev xs) = sum (sort xs)
```

Since neither side of (†) is a single call to a structurally recursive function, this equation does not fall into (F0), and indeed, the induction over it will get stuck.

To see this point, let us still consider the `x = cons h t` case, where the inductive hypothesis (IH) is as follows, which we have seen in Sect. 1.

```
sum (rev t) = sum (sort t)
```

By rewriting and reducing the original proposition with `x = cons h t`, we get the following equation.

```
sum (snoc h (rev t)) = sum (ins h (sort t))
```

Unfortunately, neither side of (IH) appears, disabling the application of the inductive hypothesis. In fact, we can formally prove that this proposition cannot be proved by only induction over the original proposition [34].

If we can transform the original proposition (†) into (F0), we can ensure to effectively apply the inductive hypothesis. One way to perform this transformation is to find an equation where one side of the equation is the same as one side of the original proposition, and the other side is a single call to a structurally recursive function. This leads to the lemma (L1), which we have seen in the introduction.

$$\forall\ \texttt{xs} : \texttt{List}.\quad \texttt{sum (rev xs)} = \texttt{sum xs} \tag{L1}$$

Rewriting (†) with (L1), we obtain (L2) we have seen.

$$\forall\ \texttt{xs} : \texttt{List}.\quad \texttt{sum xs} = \texttt{sum (sort xs)} \tag{L2}$$

Now the original proof goal (†) splits into (L1) and (L2), both conforming to (F0). Now we have the guarantee that the inductive hypothesis can be applied in the inductive proofs of both (L1) and (L2).

Automation. Most steps of the above transformation process can be easily automated, and the only difficult step is to find a suitable lemma. Based on the form of the lemma, the key is finding the structurally recursive function `sum` to be used on the RHS, equivalent to a known term `sum` ∘ `rev` on the LHS. In general, synthesizing a function from scratch may be difficult. However, synthesizing a structural recursion is significantly easier for the following two reasons. First, the template fixes a large fraction of codes in a structural recursion. In this example, the structural recursion over `xs` with the following template.

```
Let f xs =
  match xs with
  | nil → base
  | cons h t → Let r = f t in comb h r
  end;
```

where the only unknown parts are *base* and *comb*. Second, we can separate the expression for each constructor as an independent synthesis task. In this example, we have the following two independent synthesis tasks for the constructors `nil` and `cons`, respectively.

$$\texttt{sum (rev nil)} = \mathit{base}$$
$$\forall\ \texttt{h t}.\quad \texttt{sum (rev (cons h t))} = \mathit{comb}\ \texttt{h}\ (\texttt{sum}\ (\texttt{rev t}))$$

Existing program synthesizers (e.g., AutoLifter [13] in our implementation) can easily solve both tasks. We get $base = 0$ and $comb\ h\ r = h + r$. Thus, `f` coincides with `sum`. An additional benefit is that a typical synthesizer requires a verifier to verify the synthesis result. Here, we can omit the verifier and rely on tests to validate the result. This does not affect the soundness of our approach since the synthesized lemma is proved recursively.

Tactic. Summarizing the above process, we obtain the first tactic. Given a proof goal that does not conform to (F0), this tactic splits it into two proof goals, both conforming to (F0). This tactic has two variants, which rewrite the LHS and the RHS, respectively. We give only the RHS version here. In more detail, given an equation $\forall \bar{x}.p_1(\bar{x}) = p_2(\bar{x})$ that does not satisfy (F0), our first tactic proceeds as follows.

Step 1. Derive a lemma template in the form of $\forall \bar{x}, p_2(\bar{x}) = f(\bar{x})$, where f is a structurally recursive function to be synthesized.
Step 2. Generate a set of synthesis problems and solve them to obtain f.
Step 3. Generate two proof goals, $\forall \bar{x}.p_1(\bar{x}) = f(\bar{x})$ and $\forall \bar{x}.f(\bar{x}) = p_2(\bar{x})$.

Overall Process. Our approach AutoProof combines any deductive solver with the two tactics to prove equivalence between functional programs. Given an equation, our approach first invokes the deductive solver to prove the equation. If the deductive solver fails to prove, we check if the equation is in an induction-friendly form and apply induction to generate new proof goals. Otherwise, we check if any tactic can be applied, and apply the tactic to generate new proof goals. Finally, we recursively invoke our approach to the new proof goals. The workflow of solving our harder example (†) is illustrated in Fig. 2.

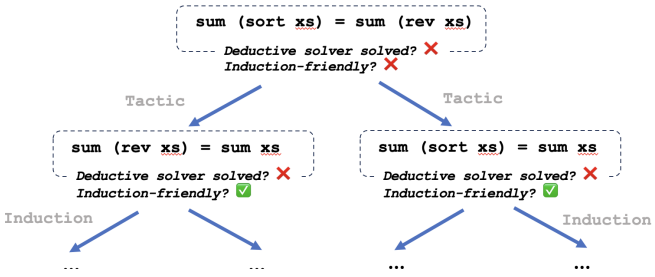

Fig. 2. Workflow of AutoProof

Towards the Full Approach. The tactic we present here attempts to transform a complex term into a single structural recursion, but it may not be possible in general. Thus, the full tactic transforms only a composition of two structural recursions into a single one each time, to significantly increase the chance of synthesis success.

Through out the section we consider only structurally recursive functions taking only one parameter, but there may be multiple ADT variables in general (e.g., proving the commutativity of natural number multiplications). Our second tactic deals with an issue caused by *inconsistent recursions*, that is, different

recursions that traverse different ADT variables. Examples and details on this tactic can be found in Sect. 4.5.

3 Preliminary

This section presents the background of program equivalence checking. We first articulate the range of equivalence checking tasks. Throughout this paper, we use $p(v_1, \ldots, v_k)$ to denote a functional program p whose free variables range from $\{v_1, \ldots, v_k\}$.

Types. The family of types in AutoProof consists of two disjoint parts: (1) the algebraic data types, and (ADTs) [28], and (2) the built-in types such as `Int` or `Bool`. For ease of presentation, we assume that there is only one built-in type `Int` for integers, and only one ADT for lists with integer elements. `List` has two constructors, `nil: List` for the empty list, and `cons: Int` $\rightarrow$ `List` $\rightarrow$ `List` that appends an integer at the head of a list. AutoProof can be easily extended to handle all ADTs and more built-in types.

Syntax. As illustrated in Fig. 3, the specification for an equivalence checking task is generated by SPEC, where each task consists of two parts.

First, a specification defines a sequence of *canonical* structural recursions (CSRs), each generated by CSRDef. A CSR f is a function whose last argument is of an ADT. It applies pattern matching to the last argument v_k, which we call the *recursive argument*, and considers all top-level constructors of v_k. If $v_k =$ `nil`, i.e., an empty list, it invokes $base(v_1, \ldots, v_{k-1})$ generated by PROG. Otherwise, $v_k =$ `cons h t`. It recursively invokes itself over `t` with all other arguments unchanged, stores the result of the recursive call in `r`, and then combines the result via the program $comb(v_1 \ldots v_{k-1}, \mathtt{h}, \mathtt{r})$ generated by PROG. The non-terminal PROG generates either a variable *var*, a numerical constant *constant*, or an application by (1) a built-in operator *op* for a built-in type (e.g., $+, -, \times$ for `Int`), (2) a constructor *ctr* of an ADT, and (3) a CSR f, followed with k programs, where k is the number of arguments required by this application.

Having defined all CSRs, a specification gives the equation $\forall \bar{x}. p_1(\bar{x}) = p_2(\bar{x})$, where p_1 and p_2 are generated by PROG.

Semantics. We adapt standard evaluation rules [1] to the syntax (Fig. 3). We defer these details to the full version [34]. We use *term reduction* to refer to a single-step evaluation.

Abstraction. An *abstraction* is a syntactic transformation from a program p to another program p' performed in steps. In each step, given a program p, it introduces a fresh variable and replaces a subprogram of p with the fresh variable. For example, we can abstract the program p of `sum (snoc (h + h) (rev t))` to p' of `sum (snoc a b)`, which replaces `(h + h)` to `a`, and `(rev t)` to `b`.

Note that if p' is an abstraction of p, any transformation on p' yields another transformation on p by simply replacing each introduced fresh variable back with the corresponding subprogram. For example, the transformation from p' to `a + (sum b)` yields the transformation from p to `(h + h) + sum (rev t)`.

$$
\begin{array}{rcl}
\mathsf{SPEC} & ::= & \mathsf{CSRDef}^* \ \forall \bar{x}.p_1 = p_2 \\
 & \text{where} & p_1, p_2 \in \mathsf{PROG} \\
\mathsf{CSRDef} & ::= & \textbf{Let } f\ v_1\ v_2\ \ldots\ v_k = \\
 & & \textbf{match } v_k \textbf{ with} \\
 & & \texttt{|\ nil} \rightarrow\ base \\
 & & \texttt{|\ cons\ h\ t} \rightarrow \textbf{Let } \texttt{r} = f\ \ v_1\ \ldots\ v_{k-1}\ \ \texttt{t} \textbf{ in } comb \\
 & & \textbf{end}; \\
 & \text{where} & base, comb \in \mathsf{PROG} \\
\mathsf{PROG} & ::= & f\ \mathsf{PROG}^* \mid ctr\ \ \mathsf{PROG}^* \mid op\ \ \mathsf{PROG}^* \mid var \mid const \\
 & \text{where} & f \text{ is a CSR, } ctr \text{ is a constructor of ADT,} \\
 & & const \text{ is a constant with the built-in type,} \\
 & & op \text{ is a primitive operator, and } var \text{ is a free variable.}
\end{array}
$$

Fig. 3. Syntax of the surface language of AutoProof.

```
1  class lemma_tactic:
2    # to be instantiated
3    def precond(eq): pass
4    def extract(eq): pass
5
6    def t_apply(eq):
7      p'_s,v = extract(eq)
8      lem = syn_lem(p'_s,v)
9      eq' = apply_lem(eq, lem)
10     return eq', lem
11 # tactics: set of built-in
12           tactics
13 def Prove(pr,eq):
14   if try_deductive(pr,eq) succeeds:
15     return
16   else:
17     if induction-friendly(eq) then:
18       subgoals = split(pr,eq)
19       for sg in subgoals: Prove(sg)
20       return
21     for t in tactics:
22       if t.precond(eq) then:
23         eq',lem = t.t_apply(eq)
24         Prove(pr,lem)
25         Prove(pr.append(lem),eq')
26         return
```

Fig. 4. Pseudocode of AutoProof

Expressivity. Compared with widely-considered structural recursions [1], CSR has two additional restrictions. First, it applies pattern-matching to only one argument. Second, it keeps other parameters unchanged in recursive calls. However, we can transform any structural recursion into a composition of CSRs by refining *defunctionalization* [8]. Thus, restricting SRs to CSRs does not affect the expressivity of functional programs, see the full version [34] for details.

4 AutoProof in Detail

4.1 The Overall Approach

The pseudo-code of AutoProof is shown in Fig. 4. The main procedure is `Prove` (Lines 11–24). The input of this procedure is a pair $(\texttt{pr}, \texttt{eq})$, termed as a *goal*, where `pr` is short for premises, which is a set of equations including all lemmas and inductive hypotheses, and `eq` is an equation denoting the current proposition to be proved. The target of a goal is to prove $\texttt{pr} \vdash \texttt{eq}$.

`Prove` wraps an underlying deductive solver responsible for performing standard deductive reasoning, such as reduction or applying a premise. `Prove` first invokes the deductive solver to prove the input goal (Line 12). If the deductive solver succeeds, the proof procedure finishes (Lines 13–14). AutoProof is compatible with any deductive solver. We choose the deductive reasoning module of the state-of-the-art solver Cvc4Ind [30] in our implementation.

Otherwise, the goal is too complex for the deductive solver to handle, which often requires finding a lemma. In this case, AutoProof first invokes `induction-friendly(e)` to check if the input equation `eq` satisfies one of the two identified forms (F1) and (F2) (defined in Sect. 4.2). If so, then by the properties of induction-friendly forms, the original goal can be split into a set of subgoals (Line 18) by induction with effective applications of the inductive hypotheses.

If not, AutoProof applies a built-in set of tactics to transform an input equation into an induction-friendly form gradually. We will discuss tactics in detail in Sect. 4.3. A tactic generally has a precondition, i.e., `precond(·)` indicating the set of applicable equations. If the tactic is applicable (Line 21), AutoProof invokes another procedure `t_apply` that synthesizes a lemma `lem` and applies this lemma to transform the input equation `eq` into another equation $\texttt{eq}'$. (Line 22). Then, `Prove` is recursively called to prove the lemma `lem` and the equation $\texttt{eq}'$ with the aid of `lem` (Lines 24–25).

In this algorithm, induction is applied only when the proof goal is in the induction-friendly form, hence we need a *progress* property that, starting from any goal, if all lemmas are successfully synthesized, the initial goal can be eventually transformed into an induction-friendly form. This property is formally proved in Theorem 3.

4.2 Induction-friendly Forms in AutoProof

AutoProof identifies two induction-friendly forms (defined at Sect. 2). Both forms guarantee the effective application of the inductive hypothesis.

(F1) The first induction-friendly form is $f\ v_1 \ldots\ v_k = p(v_1, \ldots, v_k)$, where

- **(F1.1)** One side of the equation is in the form $f\, v_1 \ldots v_k$, where f is a CSR and $v_1 \ldots v_k$ are different. From the definition of CSR, f applies pattern-matching on v_k.
- **(F1.2)** The other side of the equation is a program $p(v_1 \ldots v_k)$ satisfies the condition as follows. If v_k appears in p, then there exists an occurrence of v_k, such that (1) v_k appears as the recursive argument of the CSR it is passed to, and (2) all other arguments in this CSR invocation do not contain v_k.

```
Let app x y =
match y with
| nil → nil
| cons h t →
    cons h (app x t)
end;
```

```
Let sapp x y z =
match z with
| nil → (sum x) + (sum y)
| cons h t → h + (sapp x y z)
end;
```

Fig. 5. More CSRs for This Section

Intuitively, (F1.1) guarantees the applicability of the inductive hypothesis, and (F1.2) guarantees that there is a common term for generalization. To be more concrete, consider proving $\forall \texttt{x}, \texttt{y}, \texttt{z}.\ \texttt{sapp x y z = sum (app (app y z) x)}$, where `app` and `sapp` are defined in Fig. 5, `app` is the list concatenation function, and `sapp` calucates the sum of three concatenated lists. Note that this equation fulfills (F1). Induction over `z` and consider the `cons` case where `z = cons h t`, the LHS can be reduced to:

$$\texttt{h} + (\texttt{sapp x y t}) \ = \ \texttt{sum}\ (\texttt{app}\ (\texttt{cons h}\ (\texttt{app y t}))\ \texttt{x})$$

Due to (F1.1), the LHS contains a single call, and due to the definition of the CSR, the recursive call must take `t` as the recursive argument and keep the other argument unchanged. Therefore, the LHS must contain `sapp x y t` as a subprogram, making the induction hypothesis applicable. Applying the induction hypothesis, we get

$$\texttt{h} + (\texttt{sum}\ (\texttt{app}\ (\texttt{app y t})\ \texttt{x})) \ = \ (\texttt{app}\ (\texttt{cons h}\ (\texttt{app y t}))\ \texttt{x})$$

Due to (F1.2), either `z` do not appear in RHS, leading to exactly the same RHS as the inductive hypothesis, or we can find an occurrence of `z` in the RHS (`app y z` in this example), such that `z` is the recursive argument and all other arguments do not contain `z`. In this case, the reduction produces the recursive call `app y t`, a common subprogram on both sides. In both cases, we can generalize this subprogram to a fresh variable, yielding an effective application.

The second form is dedicated to our tactics. We propose this form to capture the lemmas proposed by our second tactic (Sect. 4.5).

(F2) The second form is $f\ v_1\ \ldots\ v_k \ = \ f'\ v'_1 \ldots\ v'_k$, where $v_i \neq v_j \wedge v'_i \neq v'_j$ for all $1 \leq i < j \leq k$, i.e., each side is a single CSR call whose arguments are distinct variables.

When the equation fulfills (F2), we can guarantee an effective application of the induction hypothesis by a nested induction over v_k and v'_k. For example, consider proving $\forall \texttt{x}, \texttt{y}, \texttt{z}.\ \texttt{sapp x y z = sapp x z y}$. We first perform induction over `z` and consider the `cons` case where $\texttt{z = cons h}_1\ \texttt{t}_1$, the goal reduces to the following equation with the hypothesis $\texttt{sapp x y t}_1 = \texttt{sapp x t}_1\ \texttt{y}$.

$$\texttt{h}_1 + \texttt{sapp x y t}_1 = \texttt{sapp x}\ (\texttt{cons h}_1\ \texttt{t}_1)\ \texttt{y}$$

Applying the hypothesis on LHS, we obtain the following subgoal:

$$\texttt{h}_1 + \texttt{sapp x t}_1\ \texttt{y} = \texttt{sapp x}\ (\texttt{cons h}_1\ \texttt{t}_1)\ \texttt{y}$$

Note that this subgoal falls into (F1), where the RHS is a single call and `y` is only used as a recursive argument, and thus an effective application of inductive hypothesis is guaranteed when we perform induction over `y`. We can see that this conformance to (F1) is guaranteed because the single call on the LHS guarantees the application of the inductive hypothesis, which will make the recursive arguments on both sides the same.

The following theorem establishes that both (F1) and (F2) are induction-friendly.

Theorem 1. *Both (F1) and (F2) are induction-friendly.*

4.3 General Routine of Tactics

In this part, we demonstrate the general routine of how tactics are applied to transform the input goal, i.e., the `t.t_apply(·)` function in Line 6 of Fig. 4. Let us start with the notation of *abstraction*.

Tactics. Informally, our tactics focus on lemmas that transform a fragment of the input equation into a single CSR invocation. Thus, it requires a subroutine `extract(·)`, which needs to be instantiated per tactic, to extract the specification of a lemma synthesis problem from the equation to be proved. The output of `extract(·)` is a tuple (p'_s, v), where p'_s is an abstraction of the subprogram to be transformed, and v is a free variable in p'_s (Line 7 in Fig. 4). The output (p'_s, v) indicates the following lemma synthesis problem.

$$\forall \tilde{v}. \forall v. f^* \ \tilde{v} \ v \ = \ p'_s(\tilde{v}, v) \qquad (\mathtt{eq}_1)$$

where $\tilde{v}$ is the set of all free variables other than v.

The approach to finding f^* has been fully presented in Sect. 2 and thus is omitted here. As long as the program synthesis succeeds in finding f^*, we propose the lemma ($\mathtt{eq}_1$) above. Since p'_s is an abstraction of some subprogram in the input equation, we can easily apply the lemma ($\mathtt{eq}_1$) to transform the input equation and obtain a new equation $\mathtt{eq}_2$ to be proved (Lines 8–9 in Fig. 4).

4.4 Tactic 1: Removing Compositions

Our first tactic is used to guarantee (F1.1). Thus, the precondition `t.precond(eq)` returns true if `eq` does not satisfy (F1.1). Below, we demonstrate the extract function in detail.

The extract function picks a non-leaf subprogram $c\ p_1\ p_2\ \ldots\ p_k$ of some side of the input equation `eq`, where c is a primitive operator, a constructor, or a CSR, $p_1 \ldots p_k$ are the arguments of c, and at least one of p_i is not a variable. Then, we abstract all arguments passed to each p_i with a fresh variable, obtaining the abstracted subprogram p'_s. We define the cost of this extraction as the number of fresh variables introduced. The extraction returns the extraction with the minimum cost. If there are several choices with the same minimum cost, we pick an arbitrary one.

For example, consider proving the equation `app (rev a) (rev (rev b)) = rev (rev (app (rev a) b))`, where `app` is the list concatenation function presented in Fig. 5. Then, we may choose the subprogram `rev (rev (app (rev a) b))` and abstract the argument `app (rev a) b` of the inner `rev` with a fresh variable `x`, obtaining p'_s = `rev (rev x)`. Since this extraction only introduces one variable, the cost is one, which is the minimum cost.

Having fixed p'_s, we then select a variable v in p'_s to be the recursive argument of the synthesized CSR f^*. We choose the variable whose corresponding lemma fulfills the maximum number of forms in (F1.1), (F1.2), and (F2). If there is a tie, we choose an arbitrary variable that reaches the maximum. Note that the lemma generated by this tactic satisfies at least (F1.1), which guarantees the applicability of the inductive hypothesis.

4.5 Tactic 2: Switching Recursive Arguments

Our second tactic is used to guarantee (F1.2), and synthesizes a lemma such as `f x y = f' y x` to switch the recursive argument of a function (recall that the recursive argument is always the last one). This tactic is only invoked when the first tactic (Sect. 4.4)

cannot apply. Thus, the precondition `precond(eq)` returns true if `eq` satisfies (F1.1) but not (F1.2). Without loss of generality, we assume the LHS is a single CSR invocation with the recursive argument x.

The extraction algorithm picks the occurrence of x with the maximum depth in the AST, where x is passed to a CSR f. Then, each p_i is either the variable x or a program that does not contain x (otherwise, we find an occurrence of x with a larger depth). We introduce fresh variables $v_1 \dots v_k$ to abstract $p_1 \dots p_k$. For some $1 \leq i < k$ such that $p_i = x$ (such i always exists since the equation violates (F1.2)), the extract outputs $p'_s = f\ v_1\ \dots\ v_k$ and $x = v_i$. Since all arguments of f are abstracted, the lemma proposed by this tactic must satisfy (F2). As a result, the lemma is induction-friendly.

For example, consider proving $\forall$`x, y, z. plus3 y z x = plus (plus x y) z`. Note that this equation satisfies (F1.1) but not (F1.2). We choose the subprogram `plus x y` and abstract it into $p'_s =$ `plus a b`. Note that `x` appears as the first argument, thus the algorithm outputs (p'_s, `a`), which requires to synthesize a lemma $\forall$`a, b. plus a b = plus' b a`. As long as the lemma is synthesized, we can replace `plus x y` to `plus' y x`, making the equation satisfying (F1.2).

4.6 Properties

First, we show the soundness of AutoProof, which is straightforward.

Theorem 2 (Soundness). *If* AutoProof *proves an input goal, then the goal is true.*

Proof. The proof of the input equation searched by AutoProof is a sequence of induction, reduction, and application of lemmas. Thus, the soundness of AutoProof follows from the soundness of these standard tactics.

Progress. As mentioned in Sect. 4.1, the effectiveness of AutoProof comes from the following progress theorem.

Theorem 3 (Progress). *Starting from any goal, if all lemmas are successfully synthesized, the initial goal can be eventually transformed into an induction-friendly form.*

5 Evaluation

We implement AutoProof on top of Cvc4Ind [30], an extension of Cvc4 with induction and the available[4] state-of-the-art prover for proving equivalence between functional programs. We choose AutoLifter [13] as the underlying synthesizer, which can solve the synthesis tasks in Sect. 4.3 over randomly generated tests. Cvc4Ind comes with a lemma enumeration module, our implementation invokes only the deductive reasoning module of Cvc4Ind. To compare the lemma enumeration with directed lemma synthesis, we evaluate AutoProof against Cvc4Ind.

Dataset. We collect 248 *standard benchmarks* from the equivalence checking subset of CLAM [12], Isaplaaner [14], and "Tons of Inductive problems" (TIP) [5], which have been widely employed in previous works [7, 12, 14, 30, 38]. We observe that these benchmarks do not consider the mix of ADTs and other theories (e.g., LIA for integer

[4] Pirate [37] is reported to have better performance than Cvc4Ind on *standard benchmarks* in our evaluation, but its code and its experimental data are not publicly accessible. Thus, we do not compare our approach against Pirate. Note that AutoProof can be combined with any deductive solver, including Pirate.

Table 1. Experimental results on the number of the solved benchmarks.

	#Solved (Standard)	#Solved (Extension)	#Solved (Total)	#Fails (Timeout)
AutoProof	**140** (↑ **16.67**%)	**21** (↑ **600**%)	**161** (↑ **30.89**%)	**109**
Cvc4Ind	120	3	123	147

Table 2. Experimental results on the average runtime.

	AvgTime(s) (Standard)	AvgTime(s) (Extension)	AvgTime(s) (Total)
AutoProof	**1.31** (↑ **97.16**%)	**3.99** (↑ **98.71**%)	**3.64** (↑ **95.47**%)
Cvc4Ind	46.13	308.58	80.36

manipulation), which is also an important fragment in practice [6,10,17–19]. Thus, we created 22 *additional benchmarks* combining the theory of ADTs and LIA by converting ADTs to primitive types in existing benchmarks, such as converting `Nat` to `Int`. Our test suite thus consists of 270 benchmarks in total.

Procedure. We use our implementation and the baseline to prove the problems in the benchmarks. We set the time limit as 360 seconds for solving an individual benchmark, the default timeout of Cvc4Ind and is aligned with previous work [7, 29, 30, 38]. We obtain all results on the server with the Intel(R) Xeon(R) Platinum 8369HC CPU, 8GB RAM, and the Ubuntu 22.04.2 system.

Results. The comparison results are summarized in Tables 1–2. Overall, AutoProof solves 161 benchmarks, while the baseline Cvc4Ind solves 123, showing that directed lemma synthesis can make an enhancement with a ratio of 30.89%. On the solved benchmarks, AutoProof takes 3.64s on average, while Cvc4Ind takes 80.36s, indicating that directed lemma synthesis can save 95.47% runtime. The results justify our motivation: compared with the directionless lemma enumeration, directed lemma synthesis can avoid wasting time on useless lemmas. Note that AutoProof shows significant strength on additional benchmarks with a mixed theory. This is because the tactics and induction-friendly forms in our approach are *purely syntactic*, making AutoProof *theory-agnostic*. In contrast, Cvc4Ind is *theory-dependent*. Thus, it is hard for Cvc4Ind to tackle benchmarks with mixed theories.

Discussion. We observe that in the failed cases, the failure to synthesize a lemma is a common cause, and this in turn is due to two reasons. The first one is that the program synthesizer fails to produce a solution for a solvable synthesis problem. For example, one equation involves an exponential function, whose implementation is extremely slow on ADT types, and the synthesizer timed out on executing the randomly generated tests. The second one is that the potential lemma requires a structural recursion that is not canonical. Though in theory such a structural recursion can be converted into compositions of CSRs, our current algorithm only supports the synthesis of CSRs, and thus cannot synthesize such lemmas. This observation shows that, if we can further improve program synthesis in future, our approach may prove more theorems.

6 Related Work

Lemma Finding in Inductive Reasoning. Due to the necessity, the lemma finding algorithm has been integrated into various architectures of inductive reasoning, including theory exploration [4, 31], superposition-based provers [7, 11, 26, 29], SMT solvers [23, 30, 36, 38], and other customized approaches [20, 32]. These approaches can be divided into two categories.

First, most of these approaches [4, 7, 11, 20, 26, 29–32, 38] apply lemma enumeration based on heuristics or user-provided templates, which often produce lemmas with little help to the proof, leading to inefficiency, as we have discussed in Sect. 1. Compared with these approaches, AutoProof considers the *directed* lemma synthesis and application, eventually producing subgoals in induction-friendly forms.

Second, there are approaches [23,36] considering the lemma synthesis over a decision procedure based on bounded quantification and pre-fixed point computation. These approaches are restricted to structural recursions without nested function invocations or constructors, which cover only 19/248 (7%) benchmarks in our test suite (Sect. 5).

Other Approaches in Functional Program Verification. There are other approaches [2,16,24,35] verifying the properties of functional programs *without* induction. These tools require the user to manually provide an induction hypothesis. Thus, these approaches cannot prove any benchmark in our test suite (Sect. 5).

Invariant Synthesis. Lemma synthesis has also been applied to verifying the properties of imperative programs [9, 15], where the lemma synthesis is often recognized as *invariant synthesis*. Since the core of imperative programs is the mutable atomic variables and arrays instead of ADTs, previous approaches for invariant synthesis [9, 15] cannot be applied to our problem. It is future work to understand whether we can extend AutoProof for verifying imperative programs.

7 Conclusion

We have presented AutoProof, a prover for verifying the equivalence between functional programs, with a novel directed lemma synthesis engine. The conceptual novelty of our approach is the induction-friendly forms, which are propositions that give formal guarantees to the progress of the proof. We identified two forms and proposed two tactics that synthesize and apply lemmas, transforming the proof goal into induction-friendly forms. Both tactics reduce lemma synthesis to a specialized class of program synthesis problems with efficient algorithms. We conducted experiments, showing the strength of our approach. In detail, compared to state-of-the-art equivalence checkers employing heuristic-based lemma enumeration, directed lemma synthesis saves 95.47% runtime on average and solves 38 more tasks over a standard benchmark set.

Acknowledgement. We sincerely thank the anonymous reviewers for their valuable feedback on this paper. This work is sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China under Grant Nos. 62161146003, the ZJNSF Major Program under grant No. LD24F020013, and the ZJU Education Foundation's Qizhen Talent program.

Data Availability Statement. The artifact in this paper is publicly available on Zenodo [33].

References

1. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. SpringerVerlag (2004)
2. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: Verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala. pp. 1–10 (2013)
3. Bradley, A.R., Manna, Z.: The calculus of computation - decision procedures with applications to verification. Springer (2007)
4. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) Automated Deduction – CADE-24. pp. 392–406. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
5. 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)
6. Codish, M., Fekete, Y., Fuhs, C., Giesl, J., Waldmann, J.: Exotic semi-ring constraints. SMT@ IJCAR **20**, 88–97 (2012)
7. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) Frontiers of Combining Systems. pp. 172–188. Springer International Publishing, Cham (2017)
8. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. p. 162–174. PPDP '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/773184.773202, `https://doi.org/10.1145/773184.773202`
9. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Ice: A robust framework for learning invariants. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 69–87. Springer (2014)
10. Gavrilenko, N., Ponce-de León, H., Furbach, F., Heljanko, K., Meyer, R.: Bmc for weak memory models: Relation analysis for compact smt encodings. In: Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31. pp. 355–365. Springer (2019)
11. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: Benzmüller, C., Miller, B. (eds.) Intelligent Computer Mathematics. pp. 123–137. Springer International Publishing, Cham (2020)
12. Ireland, A., Bundy, A.: Productive Use of Failure in Inductive Proof, pp. 79–111. Springer Netherlands, Dordrecht (1996). https://doi.org/10.1007/978-94-009-1675-3_3, `https://doi.org/10.1007/978-94-009-1675-3_3`
13. Ji, R., Zhao, Y., Xiong, Y., Wang, D., Zhang, L., Hu, Z.: Decomposition-based synthesis for applying divide-and-conquer-like algorithmic paradigms. ACM Trans. Program. Lang. Syst. (feb 2024). https://doi.org/10.1145/3648440, `https://doi.org/10.1145/3648440`, just Accepted
14. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 1. pp. 291–306. Springer (2010)
15. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. Proceedings of the ACM on Programming Languages **2**(POPL), 1–33 (2017)

16. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International conference on logic for programming artificial intelligence and reasoning. pp. 348–370. Springer (2010)
17. Lopes, N.P., Monteiro, J.: Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. International Journal on Software Tools for Technology Transfer **18**, 359–374 (2016)
18. Luick, D., Kolesar, J., Antonopoulos, T., Harris, W.R., Parker, J., Piskac, R., Tromer, E., Wang, X., Luo, N.: Zksmt: A vm for proving smt theorems in zero knowledge. Cryptology ePrint Archive (2023)
19. McCarthy, J.: Towards a mathematical science of computation. In: Program Verification: Fundamental Issues in Computer Science, pp. 35–56. Springer (1993)
20. Milovančević, D., Kunčak, V.: Proving and disproving equivalence of functional programming assignments. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023). https://doi.org/10.1145/3591258, `https://doi.org/10.1145/3591258`
21. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
22. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25. pp. 378–388. Springer International Publishing, Cham (2015)
23. Murali, A., Peña, L., Blanchard, E., Löding, C., Madhusudan, P.: Model-guided synthesis of inductive lemmas for fol with least fixpoints. Proc. ACM Program. Lang. **6**(OOPSLA2) (oct 2022). https://doi.org/10.1145/3563354, `https://doi.org/10.1145/3563354`
24. Murali, A., Peña, L., Jhala, R., Madhusudan, P.: Complete first-order reasoning for properties of functional programs. Proc. ACM Program. Lang. **7**(OOPSLA2) (oct 2023). https://doi.org/10.1145/3622835, `https://doi.org/10.1145/3622835`
25. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (oct 1979). https://doi.org/10.1145/357073.357079, `https://doi.org/10.1145/357073.357079`
26. Passmore, G., Cruanes, S., Ignatovich, D., Aitken, D., Bray, M., Kagan, E., Kanishev, K., Maclean, E., Mometto, N.: The imandra automated reasoning system (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. pp. 464–471. Springer International Publishing, Cham (2020)
27. Paulson, L.C.: Isabelle: The next 700 theorem provers (2000)
28. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software foundations. Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html p. 16 (2010)
29. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: Fontaine, P. (ed.) Automated Deduction – CADE 27. pp. 477–494. Springer International Publishing, Cham (2019)
30. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 80–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
31. Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 125–148. Springer International Publishing, Cham (2021)

32. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 407–421. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
33. Sun, Y., Ji, R., Fang, J., Jiang, X., Chen, M., Xiong, Y.: Artifact for FM Paper: Proving Functional Program Equivalence via Directed Lemma Synthesis (Jun 2024). https://doi.org/10.5281/zenodo.12532389, https://doi.org/10.5281/zenodo.12532389
34. Sun, Y., Ji, R., Fang, J., Jiang, X., Chen, M., Xiong, Y.: Proving functional program equivalence via directed lemma synthesis (2024), https://arxiv.org/abs/2405.11535
35. Vazou, N.: Liquid Haskell: Haskell as a theorem prover. University of California, San Diego (2016)
36. VK, H.G., Shoham, S., Gurfinkel, A.: Solving constrained horn clauses modulo algebraic data types and recursive functions. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022)
37. Wand, D.: Superposition: types and induction. Ph.D. thesis, Saarland University (2017)
38. Yang, W., Fedyukovich, G., Gupta, A.: Lemma synthesis for automating induction over algebraic data types. In: Schiex, T., de Givry, S. (eds.) Principles and Practice of Constraint Programming. pp. 600–617. Springer International Publishing, Cham (2019)