

Inductive Proof Automation for Coq

Sean Wilson* Jacques Fleuriot Alan Smaill

School of Informatics, The University of Edinburgh
{sean.wilson, jacques.fleuriot, a.smaill}@ed.ac.uk

Abstract

We introduce inductive proof automation for Coq that supports reasoning about inductively defined data types and recursively defined functions. This includes support for proofs involving case splits and situations where multiple inductive hypotheses appear in step case proofs. The automation uses the rippling heuristic to control rewriting in step case proofs and uses heuristics for generalising goals. Additionally, the automation caches lemmas found during proof attempts so that these lemmas may be reused in future proofs. We show that the techniques we present provide a high-level of automation for inductive proofs that improves upon what is already available in Coq. We also discuss a technique that, by inspecting finished proofs, can identify and then remove irrelevant subformulae from cached lemmas, making the latter more reusable. Finally, we compare our work to related research in the field.

1 Introduction

Induction is a common tool for reasoning about inductively defined data structures and recursively defined functions. It is well known that providing automation for inductive proofs is challenging since the latter is generally undecidable and the failure of cut elimination means most inductive proofs require new lemmas to be speculated [6]. Moreover, the search space for inductive proofs is generally large because there are choices available regarding the inductive schemas to use and the variables on which to perform induction on.

In this paper, we present inductive proof automation for Coq [4]. As developments in this system frequently require reasoning about inductive data types and recursively defined functions, inductive proof automation should make theorem proving in Coq more practical. Our automation includes the use of heuristics for generalising goals [1, 3, 5] and the rippling heuristic [7] is used for guiding step case proofs (we give a brief introduction to rippling in §2). The contributions of our work are as follows:

1. We describe the implementation of modular rippling-based inductive proof automation for Coq that is designed to support working with new data types and function definitions (see §3). This includes discussion on the rationale of our design based on our experiences of automating inductive proofs, as well as some implementation details specific to Coq.
2. We describe an approach for identifying and removing superfluous assumptions from proofs so that proofs that are cached as lemmas are more general and thus more reusable (see §3.8). The method we describe, which we call *delayed generalisation*, involves inspecting finished proofs to determine which assumptions were never used.

*This research was supported by an EPSRC DTA studentship.

3. We show that our work provides significant inductive proof automation that improve upon what is currently available in Coq (see §4). We then compare our proof automation to related research in the field. For example, our work differs from IsaPlanner in that the former is able to conjecture lemmas that include implications.

2 A Brief Introduction to Rippling

We begin with a practical introduction to rippling aimed at those unfamiliar with the heuristic. A formal, more detailed presentation of this technique can be found elsewhere [7]. Rippling can be used to guide proofs whenever a theorem (labelled the *given*) shares syntactic similarities with the conclusion. A rippling proof aims to reduce syntactic differences between the conclusion and the given so that the given can then be utilised to prove the goal.

Rippling can be used to guide the step case of inductive proofs as the inductive hypothesis typically shares syntactic similarities with the conclusion. For example, consider proving $\forall x y, \text{rev } (x ++ y) = \text{rev } y ++ \text{rev } x$ by induction on x using the standard inductive scheme for lists, where $++$ and rev denote standard functional definitions of list append and list reversal respectively. The step case goal of this proof has the following form:

$$(H : \forall y, \text{rev } (t ++ y) = \text{rev } y ++ \text{rev } t) \vdash \text{rev } ((h :: t) ++ y) = \text{rev } y ++ \text{rev } (h :: t)$$

The given here is taken as the inductive hypothesis H . The given is syntactically similar to conclusion in that, if we remove certain terms from the latter, the given will match against the conclusion. We can *annotate* which terms in the conclusion are different to the given by shading-in those terms as follows:

$$\text{rev } (h :: t) ++ y = \text{rev } y ++ \text{rev } (h :: t)$$

The shaded terms that represent differences are known as *wave-fronts*. Intuitively, annotations are correct when deleting the wave-fronts from the annotated term results in a term that matches the given. When we can annotate the conclusion in this way, we say the given *embeds* into the conclusion. The aim of the rippling proof is to find a sequence of rules to remove wave-fronts from the conclusion or move the wave-fronts to the top of term tree of the conclusion so that the given can be used to advance the proof.

When an occurrence of the given appears in the conclusion, this occurrence can be replaced with `True`. This is called *strong fertilisation*. An alternative to this is *weak fertilisation*, where the given can be used to rewrite the conclusion when the given is an equation. To determine if a proof step brings the goal closer to the stage of allowing fertilisation, we make use of a metric called a *ripple measure*. For example, the sum of distances measure involves summing the distance of each wave-front from the top of the term tree [10]. Rippling only allows the conclusion to be modified when the given still embeds into the modified conclusion and the ripple measure of the modified conclusion is better than the ripple measure of the previous conclusion. As ripple measures can only be reduced a finite number of times, rippling always terminates [7]. A rippling proof can involve the use of the same equation in both directions and rippling can control the use of, for example, associativity and commutativity rules.

We now present a rippling proof for the step case goal described above, where we re-annotate the conclusion after the application of each rewrite rule used:

$$\begin{array}{lcl}
\text{rev (h :: t) ++ y} & = & \text{rev y ++ rev (h :: t)} \\
\Downarrow & & \text{LHS rewritten by } \forall h \times y, \text{ rev ((h :: x) ++ y) = rev (x ++ y) ++ [h]} \\
\boxed{\text{rev (t ++ y) ++ [h]}} & = & \text{rev y ++ rev (h :: t)} \\
\Downarrow & & \text{RHS rewritten by } \forall h \times y, \text{ rev ((h :: x) ++ y) = rev (x ++ y) ++ [h]} \\
\boxed{\text{rev (t ++ y) ++ [h]}} & = & \text{rev y ++ } \boxed{\text{rev t ++ [h]}} \\
\Downarrow & & \text{RHS rewritten by } \forall x \times y \times z, x ++ (y ++ z) = (x ++ y) ++ z \\
\boxed{\text{rev (t ++ y) ++ [h]}} & = & \boxed{\text{rev y ++ rev t ++ [h]}} \\
\Downarrow & & \text{by } \forall x \times y \times z, x = z \rightarrow x ++ y = z ++ y. \\
\text{rev (t ++ y)} & = & \text{rev y ++ rev t}
\end{array}$$

Notice that each rule application moves the wave-fronts incrementally towards the top of the term tree. The final step removes the wave-fronts completely which then allows strong fertilisation to take place. If we had not able to strong fertilize the conclusion and we could find no more measure reducing rules to apply, we say the proof is *blocked*. There are several useful approaches for discovering lemmas that can be used to unblock rippling proofs, where the most commonly applicable is *lemma calculation* [7]. Lemma calculation involves weak fertilising the conclusion, generalising the goal and then proving this new goal with another inductive proof.

3 Inductive Proof Automation for Coq

We now describe our rippling-based inductive proof automation for Coq. The automation is composed of several modular tactics and we summarise the broad purpose of each in what follows. The `simplify` tactic (see §3.3) attempts to simplify the current goal. The `trivial` tactic (see §3.4) tries to prove the current goal outright, failing otherwise. The `induction` tactic (see §3.5) begins an inductive proof by choosing a variable and inductive scheme to perform induction with. The `ripple` tactic (see §3.6) automatically identifies assumptions that embed into the conclusion and succeeds if it can strong or weak fertilize with all embeddable assumptions. The `generalise` tactic (see §3.7) attempts to generalise the current goal. The `check` tactic succeeds when it cannot find a counterexample to the current goal. This tactic, based on the approach used by QuickCheck [9], is similar to counterexample checker tools available in other proof assistants, where the variables in the goal are instantiated with randomly generated terms and the goal is then checked for a contradiction. The implementation of this tactic is documented elsewhere [16]. A single top-level tactic controls the use of the tactics summarised above to provide inductive proof automation. We describe this top-level tactic in the next section.

3.1 Top-Level Tactic Description

The top-level tactic makes use of the Boyer-Moore theorem prover waterfall approach to structure tactic calls [5]. Any subgoal generated by `induction` is processed from the top of the waterfall. The rationale of the ordering of the tactic calls is as follows: rippling should be used to guide the proof when embeddings are present; when there are no embeddings, the goal should be simplified and a trivial proof attempted; when a trivial proof fails, the goal usually requires an inductive proof, where generalising the goal beforehand typically makes the inductive proof easier. The top-level tactic thus performs the following steps for each goal:

1. If an assumption embeds into the conclusion, the following steps are performed:
 - (a) The `simplify` and `trivial` tactics are invoked in an attempt to discharge the goal trivially, where any changes made to the goal are undone on failure. When a proof

is found here for a step case goal, this can indicate that induction was performed unnecessarily and that only case analysis was needed.

- (b) The `ripple` tactic is invoked, with backtracking occurring if `ripple` fails to fertilise the conclusion.
2. The `simplify` and `trivial` tactics are invoked.
 3. The `generalise` tactic is invoked, with backtracking taking place if an overgeneralisation is detected by the `check` tactic. If the proof after this point fails, we allow backtracking to the point before `generalise` was invoked for cases where an overgeneralisation went undetected.
 4. The `induction` tactic is invoked, with the top-level tactic being called on each subgoal generated. Subgoals that contain embeddings are processed first because, as `ripple` must fertilise the goal before induction is performed again, we find this limits unproductive proof search.

Goals are processed in a depth-first-search manner, with the top-level tactic taking a parameter that limits the number of times the `induction` tactic can be invoked on a sequence of subgoals to prevent looping.

As an example of how a typical proof is automated with only basic definitions, the proof of $\forall x y, \text{rev } (x ++ y) = \text{rev } y ++ \text{rev } x$ (from §2) first proceeds by induction `x`. The base case is simplified and generalised to $\forall x, x = x ++ []$ and discharged with a simple inductive proof. In the step case of the top-level goal, rippling manages to perform weak fertilisation. This goal is then generalised to $\forall x y z, (x ++ y) ++ z = x ++ (y ++ z)$ and discharged by a simple inductive proof on `x`. The latter steps implement the lemma calculation step of rippling.

3.2 Lemma Caching

An important feature of our automation involves the caching of proven theorems for reuse in other proof attempts. Several of our design decisions are based on generating general and thus reusable lemmas during proof search. All goals proven with the `induction` tactic are cached, where the `trivial`, `ripple` and `simplify` use these cached lemmas in different ways:

- The `trivial` tactic will try to prove goals outright using any lemmas cached so far. This improves efficiency by avoiding proof search when a goal is an instance of a theorem that has already been proven.
- The `ripple` tactic makes use of all equational cached lemmas when performing rippling proof steps (see §3.6.1). Rippling is able to productively use any rule that can reduce differences in rippling proofs, where suitable rules can increase proof coverage. As rippling always terminates [7] we do not need to be concerned that some combination of cached lemmas might lead to non-terminating behaviour.
- The `simplify` tactic performs exhaustive rewriting with equations that are hand selected as simplification rules. We are current investigating ways in which suitable lemmas can be selected automatically. A useful heuristic we have found is to select any cached lemma of the form $s = t$ as a left to right simplification rule if t is a ground term (e.g. $\forall x, x * 0 = 0$, $\forall x, \text{min } x 0 = 0$) or when t embeds into s when first-order rippling annotations are used (e.g. $\forall x, \text{rev } (\text{rev } x) = x$, $\forall f a, \text{length } (\text{map } f a) = \text{length } a$). Especially for the latter heuristic, this method identifies many useful simplification rules that are typically selected by hand.

3.3 The `simplify` tactic

We now describe the design of the tactics employed by the top-level tactic. The `simplify` tactic applies the following steps in sequence and repeats until no progress is made:

Normalisation: The goal is normalised.

Injectivity: Equational assumptions are simplified using the knowledge that type constructors are injective functions. For example, given $H : \text{cons } h \ t = \text{cons } 0 \ \text{nil}$, we can generate the assumptions $h = 0$ and $t = \text{nil}$ and discard H . We make use of Coq’s `injection` tactic for this [4].

Case splitting: To simplify conditional statements, we identify terms of the form `match` $x \dots$ and destructure x . For example, x could have type `bool` or be a disjoint sum. This step can allow further reductions to take place, producing more `match` constructs, which can again be processed by this step and so on in a nonterminating loop. To avoid looping, we simply apply a small upper limit to the number of nested case splits that can occur.

Substitution: For each assumption of the form $H : x = t$, where x is a variable and x is not a subterm of term t (i.e. a non-recursive equation), we replace all occurrences of x by t and discard H .

Use equational assumptions: For every equational assumption H , we attempt to rewrite the goal by H from left to right and, if no matches are found, from right to left. If H is used to rewrite the goal, we discard H . This step can be unsafe but we find it is generally more useful than not.

Rewriting: The goal is exhaustively rewritten with cached lemmas that have been selected for use as simplification rules (see §3.2). A conditional rewrite rule can only be used when the subgoals it generates are discharged by the `trivial` tactic. We have not implemented any automated analysis to detect when rules added by the user will cause nontermination.

3.4 The `trivial` tactic

The `trivial` tactic performs the following steps in sequence:

Lemma cache: If the goal matches any previously cached lemma, the lemma is used to prove the goal. The symmetry property of equality is used so that, for example, a lemma of the form $s = t$ will prove the goal when the conclusion has the form $t = s$.

Decision procedures: We make use of Coq’s intuitionistic propositional logic decision procedure to attempt to prove the goal. Other decision procedures could be invoked here, such as one for Presburger arithmetic.

Impossible Cases: To identify assumptions that have uninhabited types, such as $H : h :: t = []$, we make use of Coq’s `inversion` tactic [4] to reason that it is impossible to construct terms of such types. This is done by calling `inversion` on each assumption in the current goal.

3.5 The `induction` tactic

To start an inductive proof, we must pick a variable on which to perform induction and select an induction scheme to use. The `induction` tactic first performs exhaustive universal introduction and then collects a list of all unique free variables used in the conclusion that are of an inductively

defined type. Induction is then performed on the first variable collected using the standard induction scheme for the type of that variable. Should backtracking occur in the top-level tactic, induction is attempted on the next variable in the list until all choices are exhausted. We find this naive approach performs well enough in practice, which agrees with what was found when IsaPlanner was developed [10].

Before performing induction, it is usually advantageous to first modify the conclusion so that as many variables as possible are universally quantified. Quantifying variables can strengthen inductive hypotheses and gives more opportunities for rippling to fertilise in step cases. The `induction` tactic therefore reintroduces as many assumptions into the conclusion as possible before performing induction. However, we make an exception of never reintroducing propositional assumptions (i.e. type **Prop**) for reasons that we explain next. Consider the following two goals, where each goal can be transformed into the other with appropriate universal introduction and reintroduction:

1. $(x:\text{nat}) \vdash \forall (y:\text{nat}), y \neq 0 \rightarrow x + y = y + x$
2. $(x:\text{nat}) (y:\text{nat}) (P:y \neq 0) \vdash x + y = y + x$

Notice that the condition $y \neq 0$ is irrelevant to proving each goal. If we attempt to prove both goals by induction on x , we find that the second one is simple to prove but the first goal is unnecessarily difficult to prove as the inductive hypothesis will contain an implication. By never reintroducing propositional assumptions before performing induction, we thus avoid complicating certain proofs when goals contain irrelevant assumptions. However, we note that, for some proofs, it will be productive to reintroduce relevant propositional assumptions into the conclusion before performing induction (see §4).

3.6 The ripple tactic

We now give a high-level overview of the `ripple` tactic, which works are follows:

1. The assumptions that embed into the conclusion that have type **Prop** (i.e. propositions) are taken as the list of givens to use in the rippling proof attempt. The restriction on the type of the assumption is used to prevent, for example, the assumption `H : list nat` (which has type **Set**) from being considered as a given. Treating such assumptions as givens is rarely useful.
2. The tactic generates all the ways that the current goal conclusion can be modified using available equational lemmas (see §3.6.1). Only modifications that reduces differences in the conclusion with respect to the list of givens are allowed. Depth-first-search is then used to explore the search space. The list of equational lemmas used is initially populated with equations generated from function definitions (see §3.6.2).
3. Fertilisation is attempted when no further difference reducing transformations can be found. There are usually choices in the way a conclusion can be weak fertilised. The general heuristic for weak fertilisation, that we use, is that the LHS of the conclusion should only be rewritten by using a given from left-to-right and the RHS of the conclusion should only be rewritten by using a given from right-to-left. When there are multiple givens, weak fertilisation only succeeds when we can weak fertilise with all givens. We do not allow backtracking on the way weak fertilisation is performed as we find the choice is typically unimportant to a proof. Givens are rarely useful after weak fertilisation and so are discarded afterwards.

3.6.1 Ripple Proof Steps

When searching for ways to transform the conclusion, we consider every way the conclusion can be modified by only rewriting one subterm. For example, given commutativity of $+$ and the conclusion is $(a + b) + c = a + b$, we would want to generate the transformations $(b + a) + c = a + b$, $c + (a + b) = a + b$ and $(a + b) + c = b + a$. We only allow conditional rewrite rules to be applied when the side-conditions can be discharged by calling `simplify` and `trivial` in sequence. For efficiency, we do not use equations from left to right if the LHS of the equation embeds into the RHS or when the LHS is a ground term. For example, the rules $\forall x, x = x + 0$ and $\forall x, 0 = x * 0$ usually only serve to increase differences in rippling proofs when used from left to right.

To check if a modification to the conclusion was difference reducing, we use the sum of distances ripple measure [10]. When there are multiple givens, a transformation is only allowed when the following holds: all the givens that embedded before still embed, the measure for at least one given has improved and the measure for the rest of the givens are no worse than before. We make use of a similar technique as IsaPlanner to control case splitting during rippling proofs [11]. Briefly, a case split is automatically performed on x whenever a modification to the conclusion results in the conclusion containing a subterm of the form `match x ...`. The case split is only allowed if the ripple measure has been reduced in each subgoal that contains an embedding. If a generated subgoal contains no embeddings, it must be discharged when `simplify` and `trivial` are invoked in sequence to continue. The rippling proof then continues within each subgoal that contains an embedding.

3.6.2 Functions and Equations

When performing rippling proof steps, we make use of equations that are generated from function definitions. For example, the standard functions `rev` and `max` can be represented with the following two equations, where each equation targets one pattern matching clause from the function definition:

```

rev_base :                               rev [] = []
rev_step :  $\forall h\ x\ y, \text{rev } ((h :: x) ++ y) = \text{rev } (x ++ y) ++ [h]$ 

max_base:  $\forall m, \text{max } 0\ m = m$ 
max_step:  $\forall m\ n, \text{max } (S\ n)\ m = \text{match } m\ \text{with } 0 \Rightarrow S\ n \mid S\ m' \Rightarrow S\ (\text{max } n\ m')\ \text{end}$ 

```

Each equation trivially follows from the function definition and is provable by reflexivity. Notice that we can use these equations from right-to-left, which can be useful in rippling proofs, where no similar transformation can be made when performing reductions on terms in Coq. We have partial automation for generating these forms of equations from functions at the moment.

3.7 The `generalise` tactic

As opposed to only making the minimal number of generalisations needed to allow a proof by induction to succeed, the `generalise` tactic is designed to generalise the current goal as much as possible so that more reusable lemmas are discovered during proof attempts. We have found that our algorithm performs well enough in practice to outweigh the concerns regarding overgeneralisations. The `generalise` tactic performs the following steps in sequence to generalise the goal:

1. Inverse functionality is used to generalising statements of the form $\forall s\ t, f\ s = f\ t$ to $\forall s\ t, s = t$. Note that curried functions will be considered for f . We do not apply

inverse functionality to functions with more than one parameter as this often causes overgeneralisations.

2. Generalisation apart is performed when the conclusion has the following form:

$$f \ s_1 \ s_2 \ \dots \ s_n = g \ t_1 \ t_2 \ \dots \ t_n$$

f and g can be any term but the terms $s_1 \ \dots \ s_n$ and $t_1 \ \dots \ t_n$ must all be equal and n must be greater than one. Generalising apart occurs by simultaneously generalising each pair from s and t that share the same index. This approach is simple to implement and, in practice, performs reasonably well on the examples we have looked at.

3. To generalise common subterms, we first generate the set of all unique subterms s that occur more than once in the conclusion. A subterm t from s is generalised in all positions in the conclusion if the following criteria are satisfied:
 - (a) The term t is not a subterm of any of the other terms from s . This restriction is used so that the largest possible common subterms are generalised.
 - (b) The type of t is not **Prop** (e.g. $\text{int} \rightarrow \text{int}$ and $\text{S } x = 1 + x$ have this type) or **Set** (e.g. nat has this type).

3.8 Delayed Generalisation

Goals sometimes contain assumptions that are not needed to complete the proof. For example, irrelevant assumptions can appear in a subgoal when case splits are performed and frequently appear in proof obligations that arise from dependently typed programs [17]. When a cached lemma includes irrelevant assumptions, the lemma is less general and thus less reusable in future proofs.

In this section, we describe an algorithm that we have named *delayed generalisation*. Given a lemma statement P and its proof t , the purpose of this algorithm is to produce a more general lemma by inspecting both P and t to identify irrelevant subformulae that can be safely removed. As an example, consider a lemma of the form $\forall x \ y, \ x \neq y \rightarrow x + y = y + x$, where the proof of this lemma did not make use of the witness for $x \neq y$. By inspecting the lemma statement and the proof, delayed generalisation can produce a more general lemma of the form $\forall x \ y, \ x + y = y + x$. This offers an alternative to eagerly guessing which assumptions are irrelevant and removing them in the middle of a proof [1, 5], which can cause overgeneralisations to occur. However, note that delayed generalisation would not, for example, be able to remove the $x \neq y$ assumption from the lemma above if the assumption had been needlessly used in some way in the proof.

We begin with a lemma statement P and its proof $t : P$. Given that P has the form $\forall (x_1:T_1) \dots (x_n:T_n), Q$, the task of delayed generalisation is to identify which universally quantified variables can be removed from the start of P and produce a more general lemma $t' : P'$ such that P' subsumes P . To understand how we can identify which variables from P should be removed, first consider the case where P is the following:

$$\forall (x \ y \ z : \text{nat}), \ y \neq 0 \rightarrow x + y + y = y + x + y$$

Notice that, to prove this theorem, we should not have to make use of z or $y \neq 0$. The following is a Coq term for t , that gives a proof for P , where the proof involves performing exhaustive universal introduction, rewriting the LHS of the conclusion using the lemma `plus_comm` and finishing with a proof by reflexivity:

```

fun (x y z : nat) (H : y ≠ 0) ⇒
  eq_ind_r (fun t ⇒ t + y = y + x + y) refl (plus_comm x y)

```

The exact meaning of each subterm in t is unimportant except to make note of a few features. Firstly, when the proof begins by exhaustive universal introduction, t begins with a sequence of λ terms, where each λ term corresponds to a universally quantified variable from P . Here, P begins with the term $\forall (x\ y\ z:\text{nat})\ (H:y \neq 0), \dots$ and so t begins with the term $\text{fun } (x\ y\ z:\text{nat})\ (H:y \neq 0) \Rightarrow \dots$. When one of the λ terms at the start of t introduces a variable that is not used in the body of t , this represents an assumption that was not required to construct the proof. In this case, variables z and H are not used in the proof and are thus superfluous to the lemma statement.

A special case to be aware of is that universally quantified variables that occur in the conclusion of P should always be retained when generating $t' : P'$. For example, consider the case where P is $\forall n, 0 * n = 0$. A standard proof t for this lemma in Coq is $\text{fun } n \Rightarrow \text{refl_equal } 0$. Notice that the variable n is free in t , yet it would be nonsensical to eliminate the corresponding variable n from P . With the previous examples in mind, the following describes the delayed generalisation algorithm:

1. We assume P has the form $\forall (x_1:T_1) \dots \forall (x_n:T_n), Q$ and t was constructed by first performing exhaustive universal introduction. Under these assumptions, term t will have the form $\text{fun } (y_1:T_1) \Rightarrow \dots \text{fun } (y_n:T_n) \Rightarrow R$.
2. P' and t' are initially taken as copies of P and t respectively. The following operation is performed on each pair (x_i, y_i) from P' and t' : if x_i is free in Q and y_i is free in R then, in P' , the subterm $(\forall (x_i:T_i), U)$ is replaced with U and, in t' , the subterm $(\text{fun } (y_i:T_i) \Rightarrow V)$ is replaced with V . Pairs are processed from the innermost to the outermost because, for example, when x_1 and x_2 are irrelevant and x_1 occurs in T_2 , x_2 must be removed first for x_1 to be identified as irrelevant.
3. P' and t' are then used to define a new lemma which, assuming some pairs were removed from these in the previous step, will be a more general version of P .

When our automation finishes constructing a proof t for a goal g by induction, t is cached as a lemma P , delayed generalisation is used on P to produce P' and then P' is used to prove g . This step is important because if P is used to prove g , P will be instantiated with all the assumptions in g , including those that have been identified as irrelevant to P . Proving the goal by P can therefore prevent delayed generalisation from identifying irrelevant assumptions in the cached proof for the top-level goal. Finally, we note that implementing delayed generalisation is fairly trivial in Coq as proofs are represented using regular Coq terms. Proofs can thus be easily inspected and manipulated with the same techniques used to write tactics.

4 Results

We now present several examples of theorems that can be automated with our top-level tactic that cannot be automated with currently available Coq tactics. The theorems we present are versions of goals that arose when conducting case studies designed to investigate the support our automation gives when verifying dependently typed programs [16]. In these case studies, we verified tail recursive programs, sorting programs and a binary adder. We found our proof automation provided significant help, proving many proof obligations that arose without assistance.

Table 1 gives a sample of theorems that can be proven by our automation. These theorems are proven only using basic definitions to indicate how useful the automation can be when verifying properties of new data types and functions. We include theorems involving natural numbers, lists and binary trees to demonstrate that our automation supports reasoning about a variety of inductively defined data types. The recursive functions in the table all have the standard structural recursive definitions. The `insertion_sort` and `insert` functions are used to implement insertion sort in the standard way, `inorder` generates a list of nodes from a binary tree using an inorder traversal, `num_nodes` returns the number of nodes in a binary tree and `count x n` returns the number of terms equal to `n` in list `x`. The proposition `list_perm x y`, which holds when list `x` is a permutation of list `y`, is defined as $\forall n, \text{count } x \ n = \text{count } y \ n$.

Note that, for each theorem labelled X1 in the table, X2 represents a lemma that was proven with induction and cached in the process of proving X1. As well as helping to explain what the proof involved, this information is intended to be indicative that caching the lemmas proven during proof search is worthwhile as the lemmas cached tend to be general and reusable. We can also see that caching can make the automation more efficient. For example, about half of the time spent proving A1 is spent proving A2. If A2 had been cached from a previous proof, the proof for A1 would just appeal to this cached lemma rather than derive the result from scratch. As seen in the table, the performance of the automation is generally good for the examples we have tried.

The theorems in the table require proofs involving higher-order functions (e.g. C1, C2), multiple inductive hypotheses (e.g. G1), case splits during rippling (e.g. D1, D2, E1, E2, F1, F2) and non-linear arithmetic (e.g. B1). From personal experience, we note that proving examples such as these by hand can be laborious, especially when multiple case splits are needed and several lemmas need to be proved, and that full automation for such theorems makes theory development easier. Further examples of theorems that can be automated are available elsewhere [16, 17]. Taking into account the caveats in the next section, we believe that it should be possible to attain a high-level of automation for many of the lemmas proven by induction in Coq’s standard list and Peano arithmetic libraries. We aim to test our hypothesis on these theories and other built-in Coq libraries in future work.

4.1 Limitations

During our case studies, we required a proof of the following theorem to show that an implementation of quicksort would terminate: $\forall x \ y \ z, \text{list_perm } (x \ ++ \ y) \ z \rightarrow \text{length } x < S (\text{length } z)$. This theorem is challenging because the step case of the inductive proof will feature a hypothesis that contains an implication. Proving such step cases usually involves piecewise fertilization [2], which we do not currently support. We note that theorem F2, where the top-level goal contains an implication, is successfully automated because the `induction` tactic performs induction in such a way that the assumption `h ≠ n` does not appear in the inductive hypothesis (see §3.5). We have so far mostly focused on automating proofs where the conclusion of the step case is an equality statement. We would like to add support for proofs where the conclusion features user-defined relations. Such an extension would likely make use of the recently improved rewriting support for working with arbitrarily relations in Coq [15]. Moreover, we have concentrated our efforts on supporting the use of recursive functions and further work is needed for proofs that involve user-defined inductive predicates.

Label	Theorem	Time taken (s)	Lemmas cached
A1	$\forall x y, \text{rev } (x ++ y) = \text{rev } y ++ \text{rev } x$	0.27	3
A2	$\forall x y z, (x ++ y) ++ z = x ++ y ++ z$	0.15	1
B1	$\forall x y z, x * S y * z = x * (z + y * z)$	0.35	4
B2	$\forall x y, x + y = y + x$	0.12	3
C1	$\forall a, \text{fold_left } \text{plus } a 0 = \text{sum } a$	0.20	3
C2	$\forall n a, \text{fold_left } \text{plus } a n = n + \text{fold_left } \text{plus } a 0$	0.12	2
D1	$\forall a, \text{length } (\text{insert_sort } a) = \text{length } a$	0.34	2
D2	$\forall x a, \text{length } (\text{insert } x a) = S (\text{length } a)$	0.12	1
E1	$\forall a, \text{list_perm } (\text{insert_sort } a) a$	2.85	3
E2	$\forall n a, \text{count } (\text{insert } n a) n = S (\text{count } a n)$	1.50	1
F1	$\forall x y, \text{list_perm } (x ++ y)(y ++ x)$	0.43	4
F2	$\forall h x y n, h \neq n \rightarrow \text{count } (x ++ h :: y) n = \text{count } (x ++ y)n$	0.27	1
G1	$\forall a, \text{length } (\text{inorder } a) = \text{num_nodes } a$	0.24	2
G2	$\forall h x y, \text{length } (x ++ h :: y) = S (\text{length } x + \text{length } y)$	0.15	1

Table 1: A sample of theorems that can be proven by our automation from only basic definitions. The above data was produced using the Coq version 8.1 development snapshot dated 07/29/2009 running on a computer with an Intel E5200 CPU and 4Gb RAM.

5 Related Work

We are unaware of any tactics in Coq that can automate theorems that require induction to be performed, which is needed to prove the examples shown in §4 with only basic definitions. We note that Coq’s `auto` tactic, which uses a Prolog-like resolution approach, can provide help for proving examples similar to those that we have looked at, but only when induction is not required and only when `auto` is supplied with carefully chosen theorems to use. In contrast, our automation requires minimal setup to be useful as well as being able to support working with new definitions. We now review some related work in other systems.

The Boyer-Moore theorem prover [5], and its successor ACL2 [12], are well known for their inductive proof automation and are likely to be able to automate theorems similar to those that we have presented. ACL2 features a complex and fine-tuned simplification tactic which is used to simplify step case proofs, in contrast to our approach based on rippling. ACL2 also uses heuristics for choosing appropriate inductive schemas and induction variables. We note that the automation in ACL2 is more cautious about making generalisations than the automation we have presented, where the former prefers to leave complex generalisations to the role of the user. ACL2 does however include heuristics for removing irrelevant assumptions during a proof, in contrast to our delayed generalisation approach which is performed at the end of the proof.

Rippling has been implemented in other systems, such as in Clam [8], NuPrl [14] and IsaPlanner [10]. We focus on the latter as this is most closely related to our system. IsaPlanner uses rippling to automate inductive proofs in a simply typed setting within a proof planning framework. This includes for rippling proofs that involve case splits [11] and multiple hypotheses, as well as lemma caching facilities, where IsaPlanner would be able to automate many of the proofs from Table 1. However, when performing lemma calculation, IsaPlanner is unable to conjecture lemmas that contain implications [11, §5.6.1]. For example, it would be unable to conjecture theorem F2, which our automation was required to do when proving theorem F1. IsaPlanner uses a similar approach to generalise common subterms as our work but lacks heuristics for generalising apart.

The Agsy proof automation tool for Agda [13] has similarities to our tool in that the former is implemented in a similar setting to Coq and automates proofs using generalisation and induction, where proofs can include case splits. However, Agsy has limited support for rewriting with equations [13, §4] and so would be unable to support proofs that rely on the controlled use

of equational lemmas made possible by rippling. The author of the tool comments that Apsy is unable to discover simple lemmas that are needed during some proofs [13, §4]. For example, Apsy needs more than basic definitions to be able to prove theorem F1, which our system is able to automate. We note that, although Apsy does not currently cache and reuse the proofs it finds, delayed generalisation could be implemented similarly in Adga.

6 Conclusions

We have described inductive proof automation for Coq that is able to automate many proofs that could not be automated in Coq before. In particular, we have found that the automation can support working with a variety of data types and functions, as well as supporting proofs that involve case splits and multiple inductive hypotheses. As inductive proofs are common in Coq's setting, we find that this improved automation makes working in Coq more practical. We identified that the current implementation of our tactics do not yet support proofs that involve user-defined inductive predicates, user-defined relations and piecewise fertilisation, which we suggest as further work.

References

- [1] Markus Aderhold. Improvements in formula generalization. In Frank Pfenning, editor, *Automated Deduction - CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2007.
- [2] Alessandro Armando, Alan Smaill, and Ian Green. Automatic synthesis of recursive programs: The proof-planning paradigm. *Autom. Softw. Eng.*, 6(4):329–356, 1999.
- [3] Raymond Aubin. *Mechanizing structural induction*. PhD thesis, The University of Edinburgh, 1976.
- [4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [5] R. S. Boyer and J. S. Moore. *A Computational Logic*. New York: Academic Press, Orlando, 1979.
- [6] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.
- [7] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
- [8] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam System. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK, 1990. Springer-Verlag.
- [9] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., September 18–21 2000. ACM Press.
- [10] Lucas Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2005.
- [11] Moa Johansson. *Automated Discovery of Inductive Lemmas*. PhD thesis, University of Edinburgh, 2009.
- [12] Matt Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
- [13] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In *TYPES*, pages 154–169, 2004.
- [14] Brigitte Pientka and Christoph Kreitz. Automating inductive specification proofs in Nuprl. *Fundamenta Informaticae*, 1(2):189 – 209, 1998.
- [15] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009.
- [16] Sean Wilson. Supporting dependently typed functional programming with proof automation and testing. University of Edinburgh, (PhD thesis in preparation).
- [17] Sean Wilson, Jacques Fleuriot, and Alan Smaill. Automation for dependently typed functional programming. To appear in: Special Issue of Fundamenta Informaticae on Dependently Typed Programming (Draft copy available from <http://homepages.inf.ed.ac.uk/s0091720/>).