Inductive Proof Automation for Coq

Sean Wilson, Jacques Fleuriot, Alan Smaill

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

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 multiple inductive hypotheses. The automation makes use of the rippling heuristic to guide step case proofs as well as heuristics for generalising goals. We include features for caching lemmas that are found during proof search, where these lemmas can be reused in future proof attempts. We show that the techniques we present provide a high-level of automation for inductive proofs which improves upon what is already available in Coq. We also discuss an algorithm 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 [7]. In this paper, we present an inductive proof automation tactic[1] for Coq [5]. As developments in Coq 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, 6] and the rippling heuristic [8] is used for guiding step case proofs (we give a brief introduction to rippling in §2). Moreover, the automation includes features for caching the lemmas it proves during proof attempts and is designed to reuse such lemmas in future proofs (see §3.2). 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 irrelevant subformulae from lemmas to produce more general and reusable lemmas (see §3.8). The method we describe, which we call *delayed generalisation*, involves inspecting finished proofs to determine which assumptions were never used.
3. We show that our work provides significant inductive proof automation that improves 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 [11] in that the former is able to conjecture lemmas that include implications.

*This research was supported by an EPSRC DTA studentship.

[1] Source code with examples available from http://homepages.inf.ed.ac.uk/s0091720/.

The tactics described in this paper have been employed in previous work in the context of supporting dependently typed programming [18]. Here, we focus on the implementation details of these tactic and, as mentioned above, detail ways in which this automation differs from current inductive proof automation systems.

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 [8]. 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 an inductive proof as the inductive hypothesis typically shares syntactic similarities with the conclusion. For example, consider proving $\forall\ \mathsf{x}\ \mathsf{y},\ \mathsf{rev}\ (\mathsf{x} ++ \mathsf{y}) = \mathsf{rev}\ \mathsf{y} ++ \mathsf{rev}\ \mathsf{x}$ by induction on x using the standard induction principle 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:

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

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:

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

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 manipulate the wave-fronts, such as by removing them or changing their positions, in such a way 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 [11]. 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 [8]. Notably, 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:

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

Notice that the rule applications move the wave-fronts incrementally towards the top of the term tree until the final step removes the wave-fronts completely. Strong fertilisation can then take place to finish the proof.

When we cannot strong fertilize in a rippling proof and there are no remaining 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* [8]. 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 induction principle 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 [10], 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 the check tactic is documented elsewhere [17]. 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 [6]. 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. Should the ripple tactic fail to fertilise the conclusion, backtracking will occur. Specifically, ripple must succeed for the next steps to be applied.
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 such goals before induction can be 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. The above tactics are implemented using a combination of OCaml and Coq's tactic language. We note that it would be an interesting exercise to attempt to reimplement this work entirely within the latter.

As an example of how a typical proof is automated with only basic definitions, the proof for the goal $\forall$ x y, rev (x ++ y)= rev y ++ rev x (from §2) first proceeds by induction on x. The base case is simplified and generalised to $\forall$ z, z = z ++ [] and discharged with a simple inductive proof. In the step case for the top-level goal, rippling only manages to perform weak fertilisation when using basic definitions (the rippling proof in §2 made use of extra rules to allow strong 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.

3.2 Lemma Caching

An important feature of our automation involves caching lemmas found during proof search and reusing these lemmas in future proof attempts. Several of our design decisions are based on generating general and thus reusable lemmas. When a goal is proven by induction, the goal and the proof found are cached as a lemma. Cached lemmas are then utilised in future proof attempts in the following 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 [8] 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 currently 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 $\mathsf{s} = \mathsf{t}$ as a left to right simplification rule if t is a ground term (e.g. $\forall\ \mathsf{x},\ \mathsf{x} * 0 = 0$, $\forall\ \mathsf{x},\ \mathsf{min}\ \mathsf{x}\ 0 = 0$) or when t embeds into s when first-order rippling annotations are used (e.g. $\forall\ \mathsf{x},\ \mathsf{rev}\ (\mathsf{rev}\ \mathsf{x}) = \mathsf{x}$, $\forall\ \mathsf{f}\ \mathsf{x},\ \mathsf{length}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{x}) = \mathsf{length}\ \mathsf{x}$) [17]. 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:

Reductions: The goal is reduced using Coq's simpl tactic which simplifies the goal by performing computations [5].

Injectivity: Equational assumptions are simplified using the knowledge that constructors are injective functions. For example, given $\mathsf{H} : \mathsf{cons}\ \mathsf{h}\ \mathsf{t} = \mathsf{cons}\ 0\ \mathsf{nil}$, we can generate the assumptions $\mathsf{h} = 0$ and $\mathsf{t} = \mathsf{nil}$ and discard H. We make use of Coq's injection tactic for this [5].

Case splitting: To simplify conditional statements, we identify terms of the form **match** x ... and destructure x when x has a non-recursively defined type such as bool.

Substitution: For each assumption of the form $\mathsf{H} : \mathsf{x} = \mathsf{t}$, 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. This is implemented with Coq's subst tactic [5].

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.

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 $\mathsf{s} = \mathsf{t}$ will prove the goal when the conclusion has the form $\mathsf{t} = \mathsf{s}$.

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 arithmatic.

Impossible Cases: When the goal contains an assumption that has an uninhabited type, such as the type $\mathsf{h} :: \mathsf{t} = []$, we can discharge the goal by reasoning that it is impossible to construct a term that has this type. We implement this using Coq's discriminate tactic [5].

3.5 The induction tactic

To start an inductive proof, we must pick a variable to perform induction on and select an induction principle 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 principle 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 [11].

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 certain assumptions into the conclusion before performing induction. 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. $(x{:}nat) \vdash \forall\ (y{:}nat),\ y \neq 0 \rightarrow x + y = y + x$
2. $(x{:}nat)\ (y{:}nat)\ (P{:}y \neq 0) \vdash x + y = y + x$

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 goal 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 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 the lemma for 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 is difference reducing, we use the sum of distances ripple measure [11]. 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 [12]. 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 equations, where each equation targets one pattern matching clause from the function definition:

```
rev_base :                      rev [] = []
rev_step :             ∀ h x, rev (h :: x) = rev x ++ [h]

max_base: ∀ m,        max 0 m = m
max_step: ∀ m n, max (S n) m = match m with 0 ⇒ S n | S m' ⇒ S (max n m') 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 generalise statements of the form $\forall\, s\ t,\ f\ s = f\ t$ to the form $\forall\, s\ t,\ s = t$. We do not apply inverse functionality to functions with more than one parameter as this often causes overgeneralisations.

2. To generalise common subterms, we first generate the set of all 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. $\mathsf{nat} \rightarrow \mathsf{nat}$ and $\mathsf{S}\ \mathsf{x} = 1 + \mathsf{x}$ have type **Prop**) and not of type **Set** (e.g. nat has type **Set**).
3. When the conclusion is an equation, a variable x is generalised apart if x occurs the same number of times on both sides of the equation and occurs at least twice on each side of the equation. Generalising apart x is performed by simultaneously replacing the leftmost occurrence of x on both sides of the equation with a fresh variable, where this process is repeated until all occurrences of x are replaced. For example, this approach will generalise apart the occurrences of x in $\mathsf{length}\ (\mathsf{x} \mathbin{++} \mathsf{x}) = \mathsf{length}\ \mathsf{x} + \mathsf{length}\ \mathsf{x}$ to $\mathsf{length}\ (\mathsf{a} \mathbin{++} \mathsf{b}) = \mathsf{length}\ \mathsf{a} + \mathsf{length}\ \mathsf{b}$.

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 a case split is performed and frequently appear in proof obligations that arise from dependently typed programs [18]. When a proof includes irrelevant assumptions, the lemma cached from this proof 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 of the form $\forall\ (\mathsf{x}_1{:}\mathsf{T}_1) \ldots\ (\mathsf{x}_n{:}\mathsf{T}_n),\ \mathsf{Q}$ and its proof $\mathsf{t} : \mathsf{P}$, the task of delayed generalisation is to identify which universally quantified variables can be removed from P to produce a more general lemma statement P' and its proof $\mathsf{t'} : \mathsf{P'}$, such that P' subsumes P.

As an example, consider the case where P is $\forall\ \mathsf{x}\ \mathsf{y},\ \mathsf{y} \neq 0 \rightarrow \mathsf{x} + \mathsf{y} = \mathsf{y} + \mathsf{x}$ and the proof t does not make use of the witness for $\mathsf{y} \neq 0$. By inspecting P and t, delayed generalisation can produce a more general lemma of the form $\forall\ \mathsf{x}\ \mathsf{y},\ \mathsf{x} + \mathsf{y} = \mathsf{y} + \mathsf{x}$. This offers an alternative to eagerly guessing which assumptions are irrelevant and removing them in the middle of a proof [1, 6], which can cause overgeneralisations to occur. However, note that delayed generalisation would not, for example, be able to remove the $\mathsf{y} \neq 0$ assumption from the lemma above if the assumption had been needlessly used in some way in the proof.

To explain how we can identify which variables in a lemma statement are irrelevant, first consider the case where P is the following:

```
∀ (x y z : nat), y ≠ 0 → x + y + y = y + x + y
```

Notice that, to prove this theorem, we should not have to make use of z or $\mathsf{y} \neq 0$. 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 $\forall\ (\mathsf{x}\ \mathsf{y}\ \mathsf{z}{:}\mathsf{nat})\ (\mathsf{H}{:}\mathsf{y} \neq 0),\ \ldots$ and so t begins with the term $\mathsf{fun}\ (\mathsf{x}\ \mathsf{y}\ \mathsf{z}{:}\mathsf{nat})\ (\mathsf{H}{:}\mathsf{y} \neq 0) \Rightarrow \ldots$.

When one of these λ terms from 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 fun n $\Rightarrow$ refl_equal 0. Notice that the variable n is not used 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) ... $\forall$ (x_n:T_n), Q and t was constructed by first performing exhaustive universal introduction. Under these assumptions, term t will have the following form: fun (y_1:T_1)$\Rightarrow$... 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 does not occur in Q and y_i does not occur in R then, in P', the subterm ($\forall$ (x_i:T_i), U) is replaced with U and, in t', the subterm (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 by induction for a goal g and this proof is cached as a lemma P (see §3.2), 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, and thus make use of, all the assumptions in g, including any that were just identified as being irrelevant by delayed generalisation. Proving the goal by P can therefore prevent delayed generalisation from identifying irrelevant assumptions in the 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 and can thus be easily 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 provides when verifying dependently typed programs [17]. 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 using only 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

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

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

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 $\forall\ \mathsf{n},\ \mathsf{count}\ \mathsf{x}\ \mathsf{n} = \mathsf{count}\ \mathsf{y}\ \mathsf{n}$.

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 proved. Full automation for such theorems therefore makes theory development easier. Further examples of theorems that can be automated are available elsewhere [17, 18].

We now discuss some of the current limitations of our automation. During our case studies, we required a proof of the following theorem to show that an implementation of quicksort would terminate: $\forall\ \mathsf{x}\ \mathsf{y}\ \mathsf{z},\ \mathsf{list\_perm}\ (\mathsf{x} \mathbin{++} \mathsf{y})\,\mathsf{z} \rightarrow \mathsf{length}\ \mathsf{x} < \mathsf{S}\ (\mathsf{length}\ \mathsf{z})$. 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 $\mathsf{h} \neq \mathsf{n}$ 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 [16]. 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.

5 Related Work

We are unaware of any tactics in Coq that can automate theorems, such as the examples given in §4, that require induction to be performed. However, we note that a Coq tool is available that is intended to make writing inductive proofs about recursive functions easier through the generation of suitable induction principles [4]. Moreover, 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 [6], and its successor ACL2 [13], 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 uses a fine-tuned simplification tactic to simplify step case proofs, in contrast to our approach based on rippling. 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. ACL2 also incorporates heuristics for chosing appropriate induction principles and induction variables.

Rippling has been implemented in systems such as Clam [9], NuPrl [15] and IsaPlanner [11]. 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 support for rippling proofs that involve case splits and multiple givens [12], as well as lemma caching facilities. IsaPlanner would be able to automate many of the proofs from Table 1. However, IsaPlanner is unable to conjecture lemmas that contain implications when performing lemma calculation [12, §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 as well as methods to remove irrelevant assumptions from proofs.

The Agsy proof automation tool for Agda [14] 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 [14, §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 Agsy is unable discover simple lemmas that are needed during some proofs [14, §4]. For example, Agsy needs more than basic definitions to be able to prove theorem F1, which our system is able to automate. We note that, although Agsy 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.

Acknowledgements: Thanks to Lucas Dixon and to the anonymous reviewers for their helpful feedback.

References

[1] Markus Aderhold (2007): *Improvements in Formula Generalization.* In: Frank Pfenning, editor: *Automated Deduction - CADE-21*, *Lecture Notes in Computer Science* 4603, Springer, pp. 231–246.

[2] Alessandro Armando, Alan Smaill & Ian Green (1999): *Automatic Synthesis of Recursive Programs: The Proof-Planning Paradigm.* *Autom. Softw. Eng* 6(4), pp. 329–356.

[3] Raymond Aubin (1976): *Mechanizing structural induction.* Ph.D. thesis, The University of Edinburgh.

[4] Gilles Barthe, Julien Forest, David Pichardie & Vlad Rusu (2006): *Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant.* In: *Procóf 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, *Lecture Notes in Computer Science* 3945, Springer-Verlag, pp. 114–129.

[5] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag.

[6] R. S. Boyer & J. S. Moore (1979): *A Computational Logic.* New York: Academic Press, Orlando.

[7] A. Bundy (2001): *The Automation of Proof by Mathematical Induction.* In: A. Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, I, chapter 13, Elsevier Science, pp. 845–911.

[8] A. Bundy, D. Basin, D. Hutter & A. Ireland (2005): *Rippling: Meta-Level Guidance for Mathematical Reasoning.* Cambridge University Press.

[9] Alan Bundy, Frank van Harmelen, Christian Horn & Alan Smaill (1990): *The Oyster-Clam System.* In: *Proceedings of the 10th International Conference on Automated Deduction*, Springer-Verlag, London, UK, pp. 647–648.

[10] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs.* In: *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, *ACM Sigplan Notices* 35.9, ACM Press, N.Y., pp. 268–279.

[11] Lucas Dixon (2005): *A Proof Planning Framework for Isabelle.* Ph.D. thesis, University of Edinburgh.

[12] Moa Johansson (2009): *Automated Discovery of Inductive Lemmas.* Ph.D. thesis, University of Edinburgh.

[13] Matt Kaufmann & J S. Moore (1997): *An Industrial Strength Theorem Prover for a Logic Based on Common Lisp.* *IEEE Transactions on Software Engineering* 23(4), pp. 203–213.

[14] Fredrik Lindblad & Marcin Benke (2006): *A Tool for Automated Theorem Proving in Agda.* *Lecture Notes in Computer Science* 3839/2006, pp. 154–169.

[15] Brigitte Pientka & Christoph Kreitz (1998): *Automating inductive Specification Proofs in Nuprl.* *Fundamenta Informaticae* 1(2), pp. 189 – 209.

[16] Matthieu Sozeau (2009): *A New Look at Generalized Rewriting in Type Theory.* *Journal of Formalized Reasoning* 2(1), pp. 41–62.

[17] Sean Wilson. *Supporting Dependently Typed Functional Programming with Proof Automation and Testing.* University of Edinburgh, (PhD thesis in preparation).

[18] Sean Wilson, Jacques Fleuriot & Alan Smaill: *Automation for Dependently Typed Functional Programming.* *To appear in: Special Issue of Fundamenta Informaticae on Dependently Typed Programming* .