

Automation for Dependently Typed Functional Programming

Sean Wilson*, Jacques Fleuriot and Alan Smaill

CISA, School of Informatics, The University of Edinburgh

Edinburgh, UK

{sean.wilson, jacques.fleuriot, a.smaill}@ed.ac.uk

Abstract. Writing dependently typed functional programs that capture non-trivial program properties, such as those involving membership, ordering and non-linear arithmetic, is difficult in current system due to lack of proof automation. We identify and discuss proof patterns that occur when programming with dependent types and detail how the automation of such patterns allow us to work more comfortably with types, particularly subset types, that capture such program properties. We describe the application of rippling, both for inductive and non-inductive proofs, and generalisation in discharging proof obligations that arise when programming with dependent types. We then discuss an implementation of our ideas in Coq with examples of practical programs that capture useful properties. We demonstrate that our proof automation is generic in that it can provide support for working with theorems involving user-defined inductive data types and functions.

Keywords: Dependent Types, Rippling, Generalisation, Automated Theorem Proving, Coq

1. Introduction

Dependent types allow programmers to capture properties about functional programs so that program errors can be detected statically. Generally, as more expressive program properties are captured, more programming errors can be detected at compile-time. However, more expressivity inevitably means that more demanding theorem proving must be performed. In this work, we describe and provide results for generic proof automation techniques that make it easier to program with dependent types that capture useful non-trivial program properties. We are interested in supporting properties such as:

Arithmetic Properties e.g. properties about the number of items in a collection and the size/height/depth of a data structure. We are particularly interested in non-linear arithmetic properties.

*This research was supported by an EPSRC DTA studentship.

Membership Properties e.g. properties about how one collection is a subcollection or a permutation of another and capturing how many of a particular item are stored in a data structure.

Ordering Properties e.g. properties about collections being sorted and whether an item has been added to a particular position in a data structure.

Program Equivalence e.g. that an optimised version of a program produces the same results as an unoptimised version.

We make use of Coq [22] and the Program extension [29] as our dependently typed programming framework to demonstrate our ideas. The contributions of this paper are as follows:

- We identify and describe distinctive common *proof patterns* that occur when programming with dependent types (see Section 3) and describe how these can be automated. Our focus is mainly on programming with subset types (see Section 2.1.2) and the automation of inductive proofs.
- Of particular interest, we identify that rippling [8] can be used to guide the proofs required when defining recursive functions that output subset types (see Section 3.5). Unlike typical rippling proofs, our use of rippling is not restricted to the subgoals of inductive proofs. We also identify the importance of generalisation [2, 4, 7] in reasoning about dependently typed programs in Section 3.7.
- In Section 4, we describe how we have provided automation for the proof patterns we identified. These tactics provide automation for proofs involving statements about user-defined inductive types and functions over those types. Such statements can then be more easily used as type indices for inductive families and in the propositional parts of subset types. In Section 5 and 6, we show how this automation allows us to work more easily with dependent types that capture useful non-trivial program properties.

2. Background

In this section, we give an overview of a style of programming with dependent types in Coq and draw attention to where proof automation is helpful in this process. We then give a brief introduction to rippling as knowledge of when rippling can be used to guide proofs will become important later.

2.1. Dependently Typed Programming in Coq

Coq is based on the Calculus of Inductive Constructions [6] (CIC), which is both a constructive logic and a dependently typed functional programming language. Terms reductions in CIC have the confluence and strong normalization properties. CIC has decidable type-checking and uses intensional equality [30]. There are many approaches to constructing dependently typed programs in Coq, some of which involve writing the computational and logical terms of a program at the same time. In this work, we find it more convenient to separate the task of writing programs and writing proofs when programming with dependent types. Fortunately, the recent Program extension to Coq allows us to do this.

2.1.1. Program and The Russell Language

Using Program [28, 29], we can first write the computational parts of dependently typed programs in a language called *Russell* and leave the construction of the required proof terms to a later time. Russell programs look like standard simply typed functional programs. After a decidable type-checking procedure, a Russell program is interpreted into a CIC term that is missing the required proof terms. These missing proof terms become *proof obligations* that the user must solve to produce a complete CIC term. We give an introduction to programming with Russell in the next few sections. Note that we make use of the standard Coq operator `<>` for “not equals” and, when using simply typed lists, `[]` for the empty list, `::` for list concatenation and `++` for list append. For ease of presentation, consider all variables used in theorem statements as being universally quantified unless otherwise stated and note that we usually omit some assumptions from proof goal examples.

2.1.2. Subset Types

One useful way to specify properties about terms from existing types in Coq is the use of the so-called *subset type*. A subset type term combines a computational term x with the propositional term P , where P is a certificate that property P holds for x . A subset type is denoted as $\{x:A \mid P\}$, where each member is a term x in type A paired with a proof of P . Typically, transparent simply typed functions are used to form the statement P to describe some property of x . For instance, the type $\{x:\text{list } A \mid \text{length } x <> 0\}$ could be used to represent all non-empty lists of type A . A term in this subset type can be constructed given a list x and a proof that $\text{length } x <> 0$. We make use of the function `proj1_sig` to extract the computational part from a subset type term.

2.1.3. Subset Type Proof Obligations

Given the usual inductive definition for simply typed lists, the following Russell program reverses the input list `a` and returns a subset type term that contains the reversed list `o` with a proof that `o` and `a` are the same length:

```
Program Fixpoint revs (A:Set) (a:list A) {struct a} :
  {o:list A | length o = length a} :=
match a with
[] => []
| h::t => (revs t)++[h]
end.
```

The `{struct a}` part of the program says that the function is defined by structural recursion on `a`; this can sometimes be inferred. The body of the program looks like a typical functional implementation of list reverse. However, notice that the program appears to return a simply typed list as a result when the output type of `revs` is a subset type. When a term in a Russell program is expected to be of type $\{x:A \mid P\}$, Russell allows the use of a term t of type A if a proof of $P[t/x]$, in the typing context of t , is supplied later. After defining a Russell program, such terms generate proof obligations which we are then prompted to solve. For the function above, the `[]` result generates the *base case proof obligation* of `[] = a -> length [] = length a`. Notice the equational assumption. Informally, a proof obligation generated from a term inside a pattern matching clause will contain an equation encoding

what the pattern matched term matched against. In this case, the term `a` was matched against the pattern `[]`. We refer to these as *pattern matching equations*.

Russell also allows the use of a term `t` of type $\{x:A \mid P\}$ when a term of type `A` was expected. In such cases, the `proj1_sig` function is used to coerce `t` to the expected type. This happens in the definition of `revs`, as the recursive call returns a subset type and yet is used as an input to the simply typed `append` function. The result for the second match clause in `revs` generates what we will refer to as a *recursive call obligation*, as a recursive call to the function being defined features in the proof obligation. This proof obligation, with the omission of some assumptions that give the types of the variables used, looks like this:

```
reverse : forall (A : Set) (a : list A), {o : list A | length o = length a}
Heq_a : h::t = a
-----
length ((proj1_sig (revs A t)) ++ [h]) = length a
```

Proving recursive call proof obligations usually involves making use of the propositional term of the subset type returned by the recursive call. This propositional term can be accessed by destructuring the result of the recursive call into its computational part `s` and propositional part `p`:

```
Heq_a : h::t = a
s : list A
p : length s = length t
-----
length (s ++ [h]) = length a
```

Notice how the proof obligation contains computational parts of the program being defined, namely the terms `++` and `[h]`. The proof for the above requires a vital lemma about how `length`, which came from the subset type being used, interacts with the `++` operator, which originated from the program. Lemmas like this can be expected when working with subset types that use simply typed functions in their implementations. Note that we have used the simply typed version of `append` in `revs` instead of a version that returns a subset type that captures the length property of `append`'s output. This style of programming avoids the effort to refine types at every level of the program and gives flexibility in that the simply typed function can be easily reused in programs that capture other properties. Automating the proofs of such theorems is therefore important when programming with subset types in this style. In particular, we wish to provide automation for proofs that arise when user-defined inductive types and simply functions over such types are used in the propositional parts of subsets types and in the implementation of programs that return subset types.

Russell also allows us to indicate that particular cases of a program will never be evaluated by marking such places with the `!` symbol:

```
Program Fixpoint head (a:list A | a <> nil) : A :=
  match a with
  | nil => !
  | h::t => h
  end.
```

Note that `(a:list A | a <> nil)` is shorthand notation for $(x:\{a:list A \mid a \neq \text{nil}\})$. The part of the program where `!` is used produces the, simplified, proof obligation of $\{a:list A \mid [] \neq a\} \rightarrow \text{False}$. These kinds of obligations usually require us to find a contradiction in the assumptions.

2.1.4. Inductively Defined Dependent Types and Proof Obligations

The following Coq definition declares a familiar inductive family, parametrised by the type A and indexed by a natural number, that represents lists of a given length:

```
Inductive vect (A : Type) : nat -> Set :=
  vnil  : vect A 0
| vcons : forall n : nat, A -> vect A n -> vect A (S n).
```

Russell programs allow the use of a term from the inductive family I with type index x when one from the inductive family I with type index y is expected. When this is done, a proof obligation is generated asking for a proof of $x = y$. Consider this example Russell program that uses `vect` to capture the length of the list returned by a function that concatenates a list of m lists of length n together:

```
Program Fixpoint vapp (A:Set) (x y:nat) (a: vect A x) (b: vect A y)
: vect A (x + y) :=
  match a with
  vnil      => b
| vcons A h t => vcons h (vapp t b)
end.
```

```
Program Fixpoint vconcat (A:Set) (m:nat) (n:nat) (a: vect (vect A m) n)
: vect A (m * n) :=
  match a with
  vnil      => vnil A
| vcons A h t => vapp h (vconcat t)
end.
```

Typically, simply typed functions are used in type indices to express program properties, such as the use of addition and multiplication in the output types of `vapp` and `vconcat`. The base case and recursive call proof obligation here are $0 = m \rightarrow 0 = n * 0$ and $S w = m \rightarrow n + n * w = n * m$ respectively. If we do not rely on domain specific knowledge, both of these theorems require inductive proofs if addition and multiplication are defined by structural recursion. In this example, a careful choice of the type indices used for the output of `vapp` and `vconcat` and/or changing which variable addition and multiplication are structurally recursive on can allow the proof obligations to reduce to trivial theorems. However, given realistic dependently typed programs, it is inevitable that fundamental lemmas about the interactions between the functions used in the type indices must be proven eventually. We wish to provide generic automation for theorems about such functions and their use in inductive families. A significant element of the automation that we propose later is based on the rippling heuristic and we give a brief overview of this reasoning technique next.

2.2. Rippling

Rippling [8] is a successful theorem proving technique that captures a common pattern of reasoning found in mathematical proofs. Rippling applies when there is a theorem, labelled the *given*, that is syntactically similar to the conclusion, labelled the *goal*. The general concept is that the proof attempt should be directed by using rules that reduce the syntactic differences between the given and goal so the given can be used to advance the proof. Rippling was originally devised to aid the automation of

inductive theorem proving. In this domain, the given is the inductive hypothesis. Applying the inductive hypothesis is usually crucial to proving the step case in inductive proofs.

Consider the step case of an inductive proof for $\text{length } (a ++ b) = \text{length } a + \text{length } b$ from a rippling perspective after applying induction on a . Note that the variable b in the given is universally quantified; all other variables are free:

$$\begin{array}{l} \text{Given:} \quad \text{forall } b, \text{ length } (t ++ b) = \text{length } t + \text{length } b \\ \text{Goal:} \quad \text{length } ((h::t) ++ b) = \text{length } (h::t) + \text{length } b \end{array}$$

The given here is syntactically similar to the goal, the only differences being the extra “ $h::$ ” terms around the occurrences of t . We say that an *embedding* exists between the given and the goal when the term tree for the given can be decorated with terms to produce a term that matches the goal [27]. The terms that must be added indicate the differences between the goal and the given. Here we show differences between the goal and the given as shaded terms:

$$\text{Annotated Goal:} \quad \text{length } ((h::t) ++ b) = \text{length } (h::t) + \text{length } b$$

We know that the differences have been correctly annotated when removing the annotated terms produces the given. Rippling guides the use of proof steps, typically rewrite rules, to reduce differences between the goal and the conclusion. Modifications of the goal are only allowed if differences are either removed or reduced with respect to the given. Rules that perform such modifications are called *wave rules*. The following proof shows how differences can be “rippled out” using the theorems $(h::t) ++ b = h::(t ++ b)$, $\text{length } (h::t) = S (\text{length } t)$, $S x + y = S (x + y)$ and $x = y \rightarrow S x = S y$, which will be labelled $w1$, $w2$, $w3$, and $w4$ respectively. After each proof step, the differences between the modified goal and the given are annotated again:

$$\begin{array}{lll} \text{length } ((h::t) ++ b) = \text{length } (h::t) + \text{length } b & \text{Original conclusion.} \\ \text{length } (h::(t ++ b)) = \text{length } (h::t) + \text{length } b & \text{LHS differences rippled out (w1).} \\ S (\text{length } (t ++ b)) = \text{length } (h::t) + \text{length } b & \text{LHS differences rippled out (w2).} \\ S (\text{length } (t ++ b)) = S (\text{length } t) + \text{length } b & \text{RHS differences rippled out (w2).} \\ S (\text{length } (t ++ b)) = S (\text{length } t + \text{length } b) & \text{RHS difference rippled out (w3).} \\ \text{length } (t ++ b) = \text{length } t + \text{length } b & \text{Final differences removed (w4).} \end{array}$$

When the goal contains a term that matches the given, this term can be replaced with `True`. This is called *strong fertilisation*. When the given is an equation and one side of the equation matches a term t in the goal, the given can be used as a rewrite rule to replace t . This is called *weak fertilisation* and is possible after the first two steps in the proof just described. When the given contains a universally quantified variable x , strong/weak fertilisation can still occur if differences are moved next to the x term in the goal. Terms like this in the given are referred to as *sinks*. Rippling allows differences to be “rippled-in” towards sink positions to allow fertilisation to occur. As differences can only be reduced a finite number of times, rippling will eventually terminate. Theorems, such as commutativity rules, are usually troublesome in automated theorem proving because they cause looping when used without

care. Such theorems can be applied without worry in both directions in rippling as not all uses of such theorems will be difference reducing.

Automating inductive proofs is difficult due to the huge number of choices in the search space. There are choices such as which variable to perform induction on, which induction scheme to use, if the original goal should be generalised before performing induction and which theorems to use during induction. Non-trivial inductive proofs usually rely on auxiliary lemmas that need to be discovered. Rippling, although not complete, offers a practical solution to some of these problems. For instance, *lemma calculation* allows missing lemmas to be discovered [19]. Briefly, when differences can no longer be reduced, a new theorem proof is started on a weak fertilised and generalised version of the current goal. When proven, this theorem is then used as a wave rule in the original proof. The technique can be used to discover to discover the `w3` lemma in the above proof.

In the domain of dependently typed programming, we find that rippling is invaluable for automating the proof obligations that arise in practice. We discuss the uses of rippling in this context later, such as in Section 3.5.

3. Proof Patterns

When writing dependently typed programs in Coq in the style discussed in Section 2.1, we observed that common proof patterns emerged when discharging proof obligations by hand. In this section, we discuss what these proof patterns are and why they arise in our domain. For each proof pattern, we describe the pre-conditions that must be met for the pattern to apply, what transformation are made to the goal and what features the modified goal has. We first describe these patterns in isolation, starting with the simplest patterns, and then discuss how these patterns interact to describe the proofs of common proof obligations.

3.1. The `simplification` Proof Pattern

A common first step when doing any theorem proving is to transform the goal into a simpler form before more complex reasoning techniques, like induction, are used. Simplifying subset type proof obligations is required to make any progress most of the time as the propositional parts of the subset types, such as equations, must be made accessible to prove the goal. Applications of functions that return subset types and references to subset type terms generally need to be eliminated from the goal. Here we describe the `simplification` pattern:

Pre-conditions: Applies to any goal but is usually performed when we expect a goal will not be in its simplest form, such as initial proof obligation goals.

Description: The basic simplification steps taken are:

1. Apply standard term reductions that simplify the goal.
2. If the goal contains a subterm with a type of the form $\tau : \{x \mid P\}$, replace all occurrences of τ with a fresh variable $f := \tau$ then destructure f into its computational part s and propositional part p . After doing this, terms that were of the form `proj1.sig τ` can then be reduced.

3. Destructure any assumptions that have non-recursively defined types such as subset types and pairs. This usually allows more term reductions to take place.
4. For each assumption of the form $x = y$, where x is not a subterm of y (a non-recursive equation), replace all occurrences of x by y in the goal and then discard x . Equations like these sometimes appear when subset types make use of equations in their propositional parts. Non-recursive equations readily appear when using pattern matching in Russell programs.
5. For each subterm t in the goal of the form `match x with ...`, destructure x to simplify t , possibly producing subgoals. Such subterms come from functions that contain conditional statements.
6. Repeat the above steps until no progress can be made.

3.2. The elementary Proof Pattern

Some goals only require the use of simple reasoning to discharge and are described by the elementary pattern:

Pre-conditions: None.

Description: The goal is proven using standard reasoning techniques such as propositional reasoning and proof by reflexivity.

3.3. The impossible case Proof Pattern

Some goals are proven by finding contradictions in the assumptions such as for the theorems $x = y \rightarrow x <> y \rightarrow \text{False}$ and $h : t = [] \rightarrow \text{False}$. Parts of Russell programs that have been marked as cases that cannot occur during evaluation generate proof obligations like this, where the conclusion is always of the form `False`. The `impossible case` pattern is described as:

Pre-conditions: Always applies to goals when the conclusion is `False`. However, some goals that follow this proof pattern, such as base cases for inductive proofs, do not have this form.

Description: A contradictory assumption can sometimes be found by reasoning about how terms of a specific type can be constructed. For instance, by reasoning about the constructors for the `list` type, it is impossible to construct a term with the type $h : t = []$. Other goals require the use of standard propositional reasoning to find a contradiction, such as finding two propositions of the form P and `not P`.

3.4. The induction Proof Pattern

Proof by induction is an essential tool when reasoning about inductively defined data types and recursively defined programs. Subset types that make use of inductively defined types naturally generate proof obligations that require induction to prove. Inductive families indexed by inductively defined types generate universally quantified equations as proof obligations that usually require induction to prove as well. The `induction` pattern is:

Pre-conditions: Induction can be applied whenever the conclusion contains a universally quantified or free variable that is of an inductively defined type.

Description: When there are several choices of variable, one must be chosen on which to perform induction on along with a suitable induction scheme. Before induction is applied, the conclusion should be transformed so that as many free variables as possible are universally quantified. This makes the inductive hypothesis/hypotheses stronger and provides extra sinks in the step case(s). Induction produces one or more base cases subgoals and one or more step case subgoals. When standard induction schemes are used, such as those generated automatically by Coq, the inductive hypothesis/hypotheses in the step case(s) are guaranteed to embed into conclusion i.e. the rippling heuristic will be applicable to such subgoals. Base cases can sometimes contain contradictory assumptions.

3.5. The recursive call Proof Pattern

This proof pattern requires some analysis before it is presented. Assume that we are defining a dependently typed function that matches the following template, where g has type $T \rightarrow T$:

```
Program Fixpoint f x1 x2 ... xn : {o:T | P o x1 x2 ... xn} :=
  match ...
  ...
  | ... => g (f y1 y2 ... yn)
```

The term $g (f y1 y2 \dots yn)$ will generate a subset type proof obligation. The conclusion of this proof obligation will have the form $P (g (proj1.sig (f y1 y2 \dots yn))) x1 x2 \dots xn$. If the $f y1 y2 \dots yn$ term is destructured into its computational term s and propositional term p and the $proj1.sig$ term is reduced, the proof obligation goal is transformed into the form:

$$\frac{p : P \ s \ y1 \ y2 \ \dots \ yn}{P (g \ s) \ x1 \ x2 \ \dots \ xn}$$

Notice that the assumption p and the conclusion contain syntactic similarities. This is because both of these terms were generated from the propositional part of the output type of function f .

3.5.1. Embeddings

In fact, it is possible for an embedding to exist between p and the conclusion. As the term P is common to the term tree of both the conclusion and the type of assumption p , an embedding exists if each argument of P in p embeds into the argument in the same position of the function P in the conclusion. The first argument, s , will always embed into $(g \ s)$ i.e. the difference is annotated as $g \ s$. Embeddings can only be expected between xn and yn , for each n , if yn is a subterm of xn .

In terms of the body of Russell function for f , this situation occurs when, for each n , f is called recursively with the argument yn being a subterm of xn . Fortunately, most structurally recursively functional programs are actually defined in this manner. As such, we can expect embeddings to occur frequently in recursive call proof obligations. The presence of an embedding is useful as rippling can then be used to find a sequence of proof steps that allow the recursive call assumption to be used. The

observation of embeddings in proof obligations is interesting as embeddings are usually only expected in the step cases of inductive proofs.

Pattern matching equations in proof obligations usually need to be substituted to reveal embeddings. Consider what happens if the input term x_n was pattern matched against the pattern a in the definition of f above. This produces a pattern matching equation $x_n = a$ in the recursive call proof obligation. If a subterm of a is used as an argument to P in assumption p and the corresponding argument in the conclusion is the variable x_n , an embedding will only exist if we substitute x_n first.

3.5.2. Example

If we reexamine the function and recursive call proof obligation shown in Section 2.1.3, we see that there is an embedding between the recursive call assumption and the conclusion if we first substitute the pattern matching equation. This annotated goal is as follows:

```
p : length s = length t
-----
length ( s ++ [h] ) = length ( h :: t )
```

Here, P would be the function $\text{fun } s \ t \Rightarrow \text{length } s = \text{length } t$ and g would be the function $\text{fun } x \Rightarrow x \text{ ++ } [h]$. The rippling proof for this example has similarities with the proof shown in Section 2.2. When a proof obligation contains multiple recursive calls, each recursive call will produce its own proof term. By the same reasoning above, we can expect these proof terms to embed into the conclusion most of the time. For instance, this situation occurs when defining functions that manipulate inductively defined trees. An example of this is shown in Section 5.3

3.5.3. Pattern Description

We now describe the `recursive call` proof pattern:

Pre-conditions: The conclusion of the goal contains a function call to a function f that returns a subset type.

Description: The result of each function call to f is destructured, reductions are applied to any `proj1_sig` terms and pattern matching equations are substituted. The resultant goal is now likely to contain an assumption that embeds into the conclusion if the goal is a recursive case proof obligation.

3.6. The rippling Proof Pattern

The rippling proof pattern is described as follows:

Pre-conditions: One or more assumptions embed into the conclusion.

Description: The rippling heuristic is used to apply difference reducing proof steps to the conclusion, such that fertilisation with the embeddable assumptions is possible. When no more difference reducing progress can be made and strong fertilisation is not possible, lemma calculation [19] can be used to conjecture required lemmas. The proofs of these new lemmas are attempted as new

goals, the conclusions of which are typically universally quantified statements about inductively defined types.

3.7. The generalisation Proof Pattern

Generalisation [2, 4, 7] is a theorem proving technique where, instead of proving the current goal g , we prove a lemma g' that is a generalised version of g and use g' to prove g . Typical ways of generalising are to generalise common subterms, generalising variables apart and removing assumptions. Somewhat counter-intuitively, g' is usually easier to prove than the more specialised version g . In some cases, generalisation is required before performing induction so that the inductive hypothesis is strong enough for a proof to be found. At other times, generalisation only reduces the search space. More general theorems are also more reusable in other proofs than less general theorems; this is relevant to proof automation as rippling becomes more powerful when given access to more general theorems. However, generalisation can be an unsafe proof step in that a provable goal can be generalised to form a new goal that is not provable. The typical approach is to make use of a counterexample generator to detect overgeneralisations. We find that in our domain opportunities for generalisation readily crop up in several places. See Section 5 for several examples of this.

3.7.1. Common Subterms in Proof Obligations

Common subterm generalisation involves identifying a set of common non-variable subterms in the conclusion and replacing these subterms by a fresh variable. When working with equations, the usual technique is to only generalise a subterm if it occurs in both the LHS and the RHS of the conclusion. We have observed that proof obligations from dependently typed programs tend to be susceptible to common subterm generalisation. Common subterms are to be expected for most functions that capture non-trivial properties:

- Subset type proof obligations contain some of the computational parts of the program being defined, as shown in Section 2.1.3. Reductions are normally possible in initial proof obligations, usually due to the presence of constants. Such reductions typically reveal common subterms that can be generalised. This pattern is also present in proof obligations generated by the use of inductive families.
- Assume that we are defining the function f with the type $\text{forall } x_1 \ x_2 \ \dots \ x_n, \ \{o:T \mid P \ o \ x_1 \ x_2 \ \dots \ x_n\}$ and the computational result of f is called s . For some function g , many useful instances of P compare the result of $(g \ s)$ to $(g \ x_1)$, $(g \ x_2)$... and/or $(g \ x_n)$. For instance, if f was a function to append lists and g was `length`, P could be the statement that the length of the output from f was the same as the sum of the lengths of the input lists. If f returns one of its inputs, such as x_1 , as a result, the generated proof obligation is likely to contain common subterms of the form $(g \ x_1)$. Such subterms are sometimes productive to generalise.

3.7.2. Removing Irrelevant Assumptions

Another form of generalisation is to remove assumptions from the goal that are not required for a proof to be found. For instance, top-level proof obligations usually contain all the input terms of the function

being defined as assumptions. Often, some of these assumptions are not relevant to the current proof obligation and can be safely discarded.

3.7.3. Pattern Description

We now describe the `generalisation` pattern:

Pre-conditions: Applies to any goal.

Description: Some form of generalisation is applied to the goal, such as replacing common subterms by fresh universally quantified variables. The goal produced will be a more general version of the original. Overgeneralisations can be detected with the use of a counterexample checker.

3.8. Combining Proof Patterns

Typical proof obligation proofs can be described by a combination of the previously identified patterns. Here we explain how these patterns interact to describe the general shape of the proofs that we want to automate:

- Simplification followed by basic reasoning techniques will discharge many non-trivial theorems. Proofs for base case proof obligations and for the base cases generated from the use of induction typically have this shape. These proofs are described by the `simplification` pattern followed by the `elementary` or `impossible cases` pattern.
- Non-trivial goals usually require the use of induction. Before performing induction, it is beneficial to have the current goal in its simplest and most general form. This process can be described by following the `simplification`, `generalisation` and then the `induction` pattern. Step cases of induction follow the `rippling` pattern. Bases cases can involve trivial proofs or will require the use of induction again.
- As recursive call proof obligations can contain embeddings if manipulated correctly, it is important not to apply generic simplification steps to these proof obligations initially. The `recursive call` pattern is followed on these proof obligations to reveal potential embeddings. If embeddings are found, the `rippling` pattern is followed.

4. Implementation

We have implemented Coq tactics that automate the proof patterns that we have described using a combination of Ltac [15], Coq's built-in tactic language, and OCaml. Depth-first-search is used when choices must be made, such as when choosing which variable to perform induction on or whether to prove the left/right side of a disjunction. The following gives a brief explanation to how each pattern was implemented:

- The `rippling` pattern is automated using an implementation of dynamic rippling [16, 17]. This tactic automatically identifies assumptions that embed into the conclusion and uses supplied rewrite

rules to modify the conclusion so that fertilisation can occur. Rewrite rules can be generated automatically from function definitions, such as `++` and `length`, for use in rippling. Lemma calculation has been implemented, which uses a combination of weak fertilisation and generalisation to conjecture auxiliary lemmas that are needed for a proof. The implementation also supports rippling proofs that require case splits.

- The `generalisation` pattern is implemented using a maximal common subterm generalisation tactic. Assumptions that are not used in the conclusion or assumptions that do not contain variables used in the conclusion are removed as being irrelevant. A quickcheck-like [11] counterexample finder is used to identify overgeneralisations. This is also useful for identifying unprovable proof obligations, such as those generated from faulty programs.
- The `impossible case` pattern makes use of Coq’s `inversion` tactic. This tactic will prove a goal when applied to an assumption `x` when `x` is impossible to construct given the type of `x` and the available constructors for that type.
- The `induction`, `recursive call`, `simplification` and `elementary` patterns are implemented using Coq’s built-in tactics that perform standard theorem proving proof steps.

5. Some Examples

Here we give delve into some practical examples of the kinds of programs and types we can now work more comfortably with. We sketch the proofs required and describe how such proofs are automated by our implementation. We draw particular attention to the applications of rippling and generalisation. All proofs examined can be fully automated by our system.

5.1. Example: Queues as List Pairs

An efficient way to represent a purely functional queue involves the use of a pair of lists [10]. The list pair $([f_1; f_2; \dots; f_n], [b_1; b_2; \dots; b_n])$ represents the queue $[f_1; f_2 \dots; f_n; b_n; \dots; b_2; b_1]$, where f_1 is considered the front of the queue. We label the first and second list of the list pair as `f` and `b` respectively. The queue is in *normal form* when either `f` is empty or `b` is empty. The queue must be in normal form and non-empty to allow dequeuing. We formalise these definitions with the following:

```
(* A queue is a pair of lists *)
Definition queue := prod (list A) (list A).
(* Describes how to convert a list pair queue to a single list queue *)
Definition queue_to_list (q : queue) := (fst q) ++ (rev (snd q)).
(* Predicate for when a queue is empty *)
Definition is_empty (q : queue) := (fst q = []) /\ (snd q = []).
(* Queues are equal when their list representation is equal *)
Definition queue_eq q1 q2 := (queue_to_list q1) = (queue_to_list q2).
(* Predicate for when a queue is in normal form *)
Definition normal_form (q : queue) := (fst q <> []) \\/ (snd q = []).
```

We now go on to specify several queue functions, where the exact order of the items in the queue returned are verified in each case. We start by defining a function that puts a queue into normal form. We use a subset type to state that the resultant queue is in normal form and maintains the queue order:

```
Program Definition normalize (q : queue) : {o : queue | normal_form o /\ queue_eq o q} :=
  match fst q with
  | nil => ((rev (snd q)), [])
  | h::t => q
end.
```

The first match clause generates a proof obligation that has a subgoal for each side of the conjunction used in the output subset type. During proof automation, we automatically unfold the basic queue definitions given previously. The first subgoal is trivial. The second subgoal is $\text{rev } x ++ [] = \text{rev } x$. This goal can be generalised to $x ++ [] = x$ and proven with a short inductive proof. Similarly, the second match clause generates two subgoals. The first subgoal is $h::t <> [] \vee !0 = []$ and requires reasoning that an assumption of the type $h::t = []$ is impossible to construct. The second subgoal is proven with reflexivity. We now define the enqueue function:

```
Program Definition enqueue (q : queue) (x : A) :
  {o : queue | queue_to_list o = (queue_to_list q) ++ [x]} :=
  (fst q, x::(snd q)).
```

The generated proof obligation simplifies to $x ++ (\text{rev } y ++ [z]) = (x ++ \text{rev } y) ++ [z]$. This can be generalised to $x ++ (y ++ z) = (x ++ y) ++ z$. This requires a short inductive proof where a choice must be made on which variable to perform induction on. The following dequeue function returns a pair containing the dequeued item and the modified queue:

```
Program Definition dequeue (q:queue | normal_form q /\ not(is_empty q)) :
  {o : A * queue | (fst o)::queue_to_list (snd o) = queue_to_list q} :=
  match fst q with
  | nil => !
  | h::t => (h, (t, (snd q)))
end.
```

The impossible case generates the proof obligation $([] <> [] \vee !0 = []) \rightarrow \text{not } ([] = [] \wedge !0 = []) \rightarrow \text{False}$ which requires propositional reasoning to solve. The step case proof obligation is proven with reflexivity. We finish by defining a function that appends the contents of one queue onto the end of another:

```
Program Definition append_queue (a:queue) (b:queue) :
  {o:queue | queue_to_list o = queue_to_list a ++ queue_to_list b} :=
  (fst a, (rev (queue_to_list b) ++ (snd a))).
```

The generated proof obligation simplifies to $w ++ \text{rev } (\text{rev } (x ++ \text{rev } y) ++ z) = (w ++ \text{rev } z) ++ x ++ \text{rev } y$. This can be generalised to $x ++ \text{rev } (\text{rev } y ++ z) = (x ++ \text{rev } z) ++ y$. This theorem is particularly challenging to prove. One proof involves performing induction on x . The step case is provable by reflexivity after rippling out and weak fertilising. The simplified base case is the statement $\text{rev } (\text{rev } y ++ z) = \text{rev } z ++ y$. By induction on y , the base case produced, when simplified, is $\text{rev } z = \text{rev } z ++ []$. This requires a short inductive proof. The step case of the inductive proof on $\text{rev } (\text{rev } y ++ z) = \text{rev } z ++ y$ looks like the following from a rippling perspective:

```
H : forall y : list h, rev (rev x ++ y) = rev y ++ x
```

```
-----
rev (rev (h::x) ++ y) = rev y ++ h::x
```

Our rippling tactic gets stuck on this goal as it can neither strong or weak fertilise after applying difference reducing steps. This is a situation where the user can help by suggesting theorems that can be used to move differences such that fertilisation can occur. One approach is to suggest the lemma $\text{rev } (h::x) ++ y = \text{rev } x ++ h::y$. This lemma can be proved automatically by a short inductive proof. A proof can then be found automatically to the unproven step case above by using this lemma from left-to-right on the LHS of the conclusion and from right-to-left on the RHS of the conclusion to produce:

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

As both differences are now in a sink position, strong fertilisation can take place. Another approach involves suggesting associativity of $++$ as a lemma and using this with the basic definition of rev to allow weak fertilisation on the LHS.

5.2. Example: Verifying Quick Reverse

This function implements an efficient way to reverse a list using an accumulator. The output type of qrev verifies that the implementation produces the same results as rev :

```
Program Fixpoint qrev' (a b: list A) {struct a} : {o : list A | o = (rev a) ++ b} :=
  match a with
  | nil => b
  | h::t => qrev' t (h::b)
  end.
```

```
Program Definition qrev (a: list A) : {o : list A | o = rev a} := qrev' a [].
```

The recursive call proof obligation for qrev' , after weak fertilisation and generalisation, requires a proof of the lemma $x ++ h::y = (x ++ [h]) ++ y$. The proof obligation for qrev requires a simple inductive proof of $x ++ [] = x$. The use of rev in the computational part of append_queue and normalize from before can be replaced by qrev to make them more efficient. Uses of qrev in proof obligations will be replaced by rev when the non-recursive equality from the proof term from qrev is used during simplification. Apart from this, the proofs previously examined about queue functions remain the same.

An interesting aspect of this definition of qrev is that proving the correctness of such a function in a simply typed setting has frequently been used to motivate the need for generalisation to strengthen the inductive hypothesis. The natural theorem to attempt first in the simply type setting is $\text{rev } x = \text{qrev } x []$. However, the more general theorem $\text{rev } x ++ y = \text{qrev } x y$ is needed to allow fertilisation in the step case, which is not obvious at first when attempting a proof of the previous statement. In this dependently typed formulation of qrev , the appropriate invariant for qrev' becomes obvious when proving the base case proof obligation.

5.3. Example: Height of a Mirrored Tree

Here we define a datatype for binary trees and a function that calculates the height of a tree:

```

Inductive tree (A : Type) : Set :=
  empty : tree A
| node  : A -> tree A -> tree A -> tree A

Fixpoint max n m {struct n} : nat :=
match n, m with
  0, _      => m
| S n', 0   => n
| S n', S m' => S (max n' m')
end.

Fixpoint height (a : tree A) : nat :=
match a with
  empty      => 0
| node v l r => S (max (height l) (height r))
end.

```

We then define a function that mirrors a tree and capture the property that the mirrored tree should have the same height as the original tree:

```

Program Fixpoint mirror (a : tree A) : {o : tree A | height o = height a} :=
match a with
  empty      => empty
| node v l r => node v (mirror r) (mirror l)
end.

```

Typical recursive functions for trees, such as those that perform searches or generate pre/in/post-order node lists, contain recursive calls that act on each subtree. Each recursive call that appears in a proof obligations will generate a proof term. The step case proof obligation for `mirror`, after following the recursive call pattern, is:

```

x_p : height x = height r
y_p : height y = height l
-----

```

```

height (node v x y) = height (node v l r)

```

Notice that the proof terms generated from the two recursive calls to `mirror` embed into the conclusion. The parts of the conclusion that are not part of either assumption are shaded in to indicate the differences. When performing rippling on such goals, the conclusion is only modified in ways where all terms that embedded before still embed. By using the definition of `height`, the goal differences can be rippled out to get:

```

S (max (height x) (height y)) = S (max (height l) (height r))

```

Domain	Theorem Proven	Time taken (s)
Linear arithmetic	$n + m = m + n$	0.74
	$(x + y) + z = x + (y + z)$	0.36
Non-linear arithmetic	$m * n = n * m$	1.30
	$n * (m + p) = n * m + n * p$	1.20
	$(n * m) * p = n * (m * p)$	0.82
	$x ^ (n + m) = x ^ n * x ^ m$	1.70
Lists	$(l ++ m) ++ n = l ++ (m ++ n)$	0.34
	$\text{length } (x ++ y) = \text{length } x + \text{length } y$	0.36
	$\text{rev } (\text{rev } x) = x$	1.10
	$\text{rev } (a ++ b) = \text{rev } b ++ \text{rev } a$	1.36
	$\text{rev } x ++ h :: y = \text{rev } (h :: x) ++ y$	0.38
	$\text{count } (y ++ x) n = \text{count } (x ++ y) n$	6.10
	$h <> n \rightarrow (\text{count } (x ++ [h]) n = \text{count } x n)$	5.10
Trees	$\text{mirror } (\text{mirror } t) = t$	0.41
	$\text{num_nodes } (\text{mirror } t) = \text{num_nodes } t$	0.57
	$\text{height } (\text{mirror } t) = \text{height } t$	0.65

Table 1. Selection of example theorems that can be fully automated with our proof automation from only basic definitions. Timings were produced on Coq version 8.1 running on an AMD 3800 X2 CPU.

The conclusion can then be weak fertilised to:

$$S (\max (\text{height } r) (\text{height } l)) = S (\max (\text{height } l) (\text{height } r))$$

This can be generalised to $\max n m = \max m n$. The proof of this requires a short inductive proof involving a case split. Generally, we find that we must prove fundamental properties of all functions that we make use when capturing non-trivial program properties. As such, we find automated inductive proofs invaluable when working with dependent types.

6. Proof Automation Results

In this section, we provide a few non-trivial examples of the kinds of theorems that can be fully automated by our rippling tactics. Table 1 gives a selection of example theorems from various domains, such as non-linear arithmetic, lists and trees, that can be proven automatically. Theorems such as these occur naturally when capturing program properties with dependent types. For instance, the power operator appears in proof obligations when capturing the number of items in a binary tree and the number of items in a powerset. List functions like append and reverse are useful for capturing properties about the order of items in a collection, such as checking items in a queue appear in the right order. Functions that count the number of items in a collection are useful for verifying that functions return permutations of their inputs, remove particular items, contain a particular item or contain no duplicate items.

Each theorem in Table 1 is proven using induction with the standard induction schemes generated by Coq. Each proof is produced using only the information from basic function definitions and with no help from additional theorems. This lack of domain specific knowledge demonstrates that the proof automation is able to work with new user-defined inductively defined data types and recursively defined

functions that use such data types. This means users can define their own types and functions for use as the propositional part of subset types and/or type indices and our proof automation can provide support for this.

Despite there being scope for optimisation in our implementation, we have found our proof automation to have satisfactory performance in experiments. Typical proofs of non-trivial theorems tend to be found quickly, as shown in Table 1. Searches for proofs of non-theorems and for theorems that cannot be proven are abandoned in similar time frames. For theorems that cannot be proven, the user is able to suggest new theorems to be proven that can allow rippling to progress. An example of this is shown in Section 5.1 for a difficult theorem involving reversing lists.

7. Related Work

In this section, we review some related work on dependently typed functional programming and proof automation. The DML [31], ATS [14] and Sage [18] languages each provide some support for automating proofs that arise when programming with dependent types. Unlike Coq, DML [31] restricts the user to type indices that form linear arithmetic statements. This is done so that generated proof obligations can be solved by a constraint solver during the type checking process. ATS [14] provides proof automation for the same kinds of statements as DML but allows user-defined types and the functionality for the user to write proofs. The use of such types when programming will usually require the user to write inductive proofs. Clearly, automated inductive theorem proving would be beneficial here. The Sage [18] language also allows user defined types but, again, provides little proof automation support for working with them. Proof obligations, generated during type checking, are translated into a form that can be processed by an external theorem prover that handles linear arithmetic statements. As with our system, a counterexample checker is employed to identify unprovable proof obligations. When Sage’s proof automation fails to verify a program property statically and a counterexample cannot be found, Sage can enforce the program property at run-time using a dynamic check. This is a useful alternative for when the user is unable to provide a proof in cases where proof automation fails.

The Cayenne [5], Epigram [23] and Agda [13] systems allow user-defined dependent types but provide little in the way of proof automation and proofs need to be written by the user for even trivial theorems. Programs in these languages tend to rely on the structure of proofs to closely follow the structure of the programs themselves to avoid the need for manual theorem proving. Proofs that are written are typically performed using induction and, as such, rippling technology would be beneficial in these systems.

The Oyster-Clam [9] and NuPRL [12] systems, which are based on Martin-Löf type theory [21], both provide rippling based automation tactics. However, rippling has not been used in these systems in the context of automating proof obligations that arise from dependently typed functional programs. In the case of NuPRL, the focus of the rippling work was the instantiation of meta-variables in inductive proofs and its use in program synthesis [26]. The Isabelle theorem prover [24] is another system that has an implementation of rippling [17]. Although there is a representation of dependent types in Isabelle for reasoning about theory modules [20], there are no facilities for writing functional programs with dependent types. We are also aware that there were plans to bring rippling to PVS [25] for use in inductive proofs [1] but an implementation of this is yet to appear. Coq [22] comes with many tactics that are useful when discharging proof obligations. Several tactics are available that can solve/simplify

goals using a database of supplied theorems and there are also useful decision procedures, such as those for Presburger arithmetic and intuitionistic propositional calculus. However, as far as we are aware, there are no tactics that provide automation for working with user-defined types that require inductive proofs. We also find that our proof automation can solve theorems about built-in data-types, such as variants of standard list and non-linear arithmetic theorems, that Coq's regular tactics fail on.

8. Conclusions

In this paper, we examined several patterns of proof that commonly emerge when discharging proof obligations that arise when working with expressive dependent types. We observed that induction, rippling and generalisation are important reasoning techniques in this domain. This included the novel observations that, if manipulated correctly, embeddings can be found in the recursive call proof obligations of programs that make use of subset types and that generalisation was particularly relevant to subset type proof obligations. We gave initial results in automating the proof patterns we identified with some practical examples of what programs and properties such automation allows one to work more comfortably with. Already, we can work with examples that are either not automated in other dependently typed systems or would require difficult and/or tedious theorem proving to be performed by hand. In future work, we plan to improve the power of our proof automation and carry out larger case studies into how such automation can aid programming with dependent types. This includes looking at how the user can interact with the proof automation when it fails and how counterexamples to unprovable proof obligations can be used to help the user correct faulty programs.

References

- [1] Adams, A. A., Dennis, L. A.: Rippling in PVS, *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)* (M. Archer, B. D. Vito, C. Munoz, Eds.), NASA Technical Report CP-2003-212448, 2003.
- [2] Aderhold, M.: Improvements in Formula Generalization., *Automated Deduction - CADE-21* (F. Pfenning, Ed.), 4603, Springer, 2007, ISBN 978-3-540-73594-6.
- [3] Altenkirch, T., McBride, C., McKinna, J.: Why Dependent Types Matter, Manuscript, available online, April 2005.
- [4] Aubin, R.: *Mechanizing structural induction (formal system)*, Ph.D. Thesis, 1976.
- [5] Augustsson, L.: Cayenne - a Language with Dependent Types, *International Conference on Functional Programming*, 1998.
- [6] Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer Verlag, 2004.
- [7] Boyer, R. S., Moore, J. S.: *A Computational Logic*, New York: Academic Press, Orlando, 1979.
- [8] Bundy, A., Basin, D., Hutter, D., Ireland, A.: *Rippling: Meta-Level Guidance for Mathematical Reasoning*, Cambridge University Press, 2005.
- [9] Bundy, A., van Harmelen, F., Horn, C., Smaill, A.: The Oyster-Clam System, *Proceedings of the 10th International Conference on Automated Deduction*, Springer-Verlag, London, UK, 1990, ISBN 3-540-52885-7.

- [10] Burton, F. W.: An Efficient Functional Implementation of FIFO Queues, *Inf. Process. Lett.*, **14**(5), 1982, 205–206.
- [11] Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs, *ACM SIGPLAN Notices*, ACM Press, 2000.
- [12] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., Smith, S. F.: *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, NJ, 1986.
- [13] Coquand, C.: The AGDA Proof System Homepage, 1998, <http://www.cs.chalmers.se/~catarina/agda/>.
- [14] Cui, S., Donnelly, K., Xi, H.: Ats: A language that combines programming with theorem proving, of *Lecture Notes in Computer Science*, Springer, 2005.
- [15] Delahaye, D.: A Tactic Language for the System Coq, *LPAR* (M. Parigot, A. Voronkov, Eds.), 1995, Springer, 2000.
- [16] Dennis, L. A., Green, I., Smaill, A.: Embeddings as a Higher-Order Representation of Annotations for Rippling, 2005.
- [17] Dixon, L.: *A Proof Planning Framework for Isabelle*, Ph.D. Thesis, University of Edinburgh, 2005.
- [18] Gronski, J., Knowles, K., Tomb, A., Freund, S. N., Flanagan, C.: Sage: Hybrid checking for flexible specifications, *In Scheme and Functional Programming Workshop*, 2006.
- [19] Ireland, A., Bundy, A.: Productive use of failure in inductive proof, *Journal of Automated Reasoning*, **16**, 1996, 16–1.
- [20] Kammüller, F.: Modular Reasoning in Isabelle, *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, Springer-Verlag, London, UK, 2000, ISBN 3-540-67664-3.
- [21] Martin-Löf, P.: A theory of types, Manuscript, 1971.
- [22] The Coq development team: *The Coq proof assistant reference manual*, LogiCal Project, 2006, Version 8.1.
- [23] McBride, C., McKinna, J.: The View from the Left, *Journal of Functional Programming*, **14**(1), 2004, 69–111.
- [24] Nipkow, T., Paulson, L. C., Wenzel, M.: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*, Springer, 2002.
- [25] Owre, S., Shankar, N., Rushby, J. M., J.Stringer-Calvert, D. W.: *PVS System Guide*, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
- [26] Pientka, B., Kreitz, C.: Instantiation of Existentially Quantified Variables in Inductive Specification Proofs, *AISC '98: Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation*, Springer-Verlag, London, UK, 1998, ISBN 3-540-64960-3.
- [27] Smaill, A., Green, I.: Higher-order annotated terms for proof search, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLS 96*, Springer, 1996.
- [28] Sozeau, M.: Program-ing Finger Trees in Coq, *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, ACM Press, 2007, ISBN 978-1-59593-815-2.
- [29] Sozeau, M.: Subset Coercions in Coq, *TYPES'06*, 4502, Springer, 2007.
- [30] Werner, B.: *Méta-théorie du Calcul des Constructions Inductives*, Ph.D. Thesis, Université Paris VII, 1994.
- [31] Xi, H., Pfenning, F.: Dependent Types in Practical Programming, *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, New York, NY*, 1999.