Supporting Dependently Typed Functional Programming with Proof Automation and Testing

Sean Wilson

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh
2011

Abstract

Dependent types can be used to capture useful properties about programs at compile time. However, developing dependently typed programs can be difficult in current systems. Capturing interesting program properties usually requires the user to write proofs, where constructing the latter can be both a difficult and tedious process. Additionally, finding and fixing errors in program scripts can be challenging.

This thesis concerns ways in which functional programming with dependent types can be made easier. In particular, we focus on providing help for developing programs that incorporate user-defined types and user-defined functions. For the purpose of supporting dependently typed programming, we have designed a framework that provides improved proof automation and error feedback.

Proof automation is provided with the use of heuristic based tactics that automate common patterns of proofs that arise when programming with dependent types. In particular, we use heuristics for generalising goals and employ the rippling heuristic for guiding inductive and non-inductive proofs. The automation we describe includes features for caching and reusing lemmas proven during proof search and, whenever proof search fails, the user can assist the prover by providing high-level hints.

We concentrate on providing improved feedback for the errors that occur when there is a mismatch between the specification of a program, described with the use of dependent types, and the behaviour of the program. We employ a QuickCheck-like testing tool for automatically identifying these forms of errors, where the counterexamples generated are used as error messages.

To demonstrate the effectiveness of our framework for supporting dependently typed programming, we have developed a prototype based around the Coq theorem prover. We demonstrate that the framework as a whole makes program development easier by conducting a series of case studies. In these case studies, which involved verifying properties of tail recursive functions, sorting functions and a binary adder, a significant number of the proofs required were automated.

Acknowledgements

Firstly, thank you to my supervisors Jacques Fleuriot and Alan Smaill for all their time, advice and encouragement over the years. I would also like to thank Lucas Dixon for his help and for initially suggesting the field of dependently typed programming to me. Thank you to my examiners Ewen Denney and Stephen Gilmore for their helpful feedback. Finally, thank you to Emer, Priya and my family for their support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Sean Wilson*)

Table of Contents

Chapter 1

Introduction

Dependent types [Martin-Löf, 1971] can be used to verify useful program properties at compile time. For example, dependent types can be used to statically verify that a program only performs safe array accesses [Xi, 1999a] and that a list sorting function always returns a sorted list [Altenkirch et al., 2005]. As well as providing an approach to reducing software faults, dependent types have been previously used as a means for performing compile time optimisations [Xi and Pfenning, 1998, Brady, 2005] and eliminating dead code from programs [Xi, 1999a].

Several dependently typed programming languages exist that support programming with what we will call *user-defined properties*. By this, we mean any program properties that are described with the use of data types or functions whose definitions were introduced by the user. Current dependently typed languages that allow programming with user-defined properties include Coq [Bertot and Castéran, 2004], Cayenne [Augustsson, 1998], Agda [Coquand, 1998], Epigram [McBride and McKinna, 2004] and ATS [Cui et al., 2005]. However, developing programs in these languages can be challenging for a number of reasons:

- To capture interesting program properties, the user is typically required to construct proofs. Proof construction can range from being simple yet tedious to complex and challenging.
- Errors occur when there is a mismatch between the specification of a program, described with the use of dependent types, and the actual behaviour of the program. With the exception of a tool available for Agda [Qiao Haiyan, 2003], there is little support in current systems for identifying or giving feedback for such errors.

1.1 Hypothesis

This thesis describes a framework designed to make dependently typed functional programming more practical. The challenges we described above are addressed by this framework through the application of proof automation and testing.

Proof automation is used to assist in the construction of any proofs required during program development. In cases where the proof automation fails, a hinting mechanism is available where the user can suggest important lemmas that should be proven before another proof search is attempted. Testing is used to provide feedback for errors and faulty hints, as well as for guiding proof search. The intended audience for this framework is existing users of dependently typed programming languages who are comfortable with writing proofs.

The hypothesis we argue in this thesis is that:

"This framework makes dependently typed programming significantly easier"

By this, we mean that 1) the framework should be able to automate a significant number of the proofs that are required when writing dependently typed programs in practice and that 2) the error feedback feature helps the user to correct errors more quickly than without this feature.

We give evidence for the above by providing an analysis of our experiences when developing programs with the whole framework, where we show that many of the proofs required can be automated.

1.2 Contributions

The primary contribution of this thesis is as follows:

> We show that by integrating ideas from the domains of proof automation and testing, we can make dependently typed functional programming more practical, specifically when working with user-defined properties.

This contribution involves two key areas:

- We present generic and modular proof automation designed to construct the proofs required when programming with dependent types. In particular, this automation is shown to provide significant support for verifying program properties concerning inductively defined types and recursively defined functions.

- We demonstrate how this proof automation can be combined with testing to create an effective tool for supporting dependently typed programming. Testing is used to give feedback to faulty programs and faulty user hints, as well as for avoiding unnecessary search during proof automation attempts.

In the course of establishing the above, we have created a concrete implementation of our ideas in the Coq theorem prover. The practical contributions of this are as follows:

- We have made significant contributions to the automation power of Coq by introducing rippling-based [Bundy et al., 2005] inductive proof automation.
- We have created a QuickCheck-like [Claessen and Hughes, 2000] testing tool for testing Coq goals, developed mostly within Coq itself. Surprisingly, we are not aware of any tools with similar functionality available in Coq.

1.3 Thesis Outline

The structure of the thesis can be summarised as follows:

Chapter 2: We describe necessary background information by summarising current dependently typed programming environments, as well as surveying automated reasoning and program testing techniques.

Chapter 3: We describe the ways in which dependent types can be employed to write programs that capture useful program properties.

Chapter 4: We explain how dependently typed programming can be challenging and identify the need for improved proof automation and error feedback.

Chapter 5: We give a high-level overview of our framework for supporting dependently typed programming.

Chapter 6: We give an analysis of what we find are the common patterns of proof that arise when programming with dependent types.

Chapter 7: We describe how effective proof automation can be provided for automating the proof patterns we identified.

Chapter 8: We describe how testing can be used to provide error feedback and explain how testing and proof automation are combined to create our framework.

Chapter 9: We evaluate the effectiveness of our framework as a whole by conducting several case studies. These case studies explore the use of a variety of data types, program properties and representations.

Chapter 10: We finish by giving the conclusions of the thesis along with a discussion of further work.

1.4 Publications

Work from this thesis has previously been published [Wilson et al., 2010a,Wilson et al., 2010b]. These publications mostly concern work from chapters 6 and 7.

Chapter 2

Background

In this chapter, we present background material and summarise the previous research that our work builds upon. We start with an overview of the general features of dependently typed programming and then describe the various development environments currently available. The framework we have designed to support dependently typed programming uses ideas from the proof automation and program testing communities. We give background information on these two domains in the later sections of this chapter.

2.1 Types and Programming

The use of types in programming languages is widely recognised as making software development more practical. Types are used in programs to describe properties about data and the behaviour of functions. Typically, this is done so that type checking can be used to detect program faults at compile time.

For instance, working within Church's simply typed lambda calculus [Church, 1940] and given that the type int represents integers, the type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ could be used to describe the behaviour of a function called plus for summing two integers. Type checking will identify the term $\mathsf{plus}\ 1\ 2$ as well-typed and reject nonsensical terms like $\mathsf{plus}\ 1\ 2\ 3$ and $\mathsf{plus}\ \mathsf{plus}$. We now note some of the important uses of types in software development, based on observations made in [Pierce, 2002]:

Safety: As shown above, type checking can be used to guarantee at compile time that certain forms of errors are absent from programs. This is known by the slogan "well-typed programs do not go wrong" [Milner, 1978].

Abstraction: Types can be used to abstract away implementation details of programs to create modular code, where modules communicate through well-defined interfaces. Modular code is widely recognised as being easier to develop, maintain and reuse.

Documentation: Types are a form of machine checked documentation. For example, the type of a function can give hints about its behaviour and purpose.

Efficiency: Machines can make use of types to improve program execution performance. For example, Fortran's [IBM, 1954] type system was introduced so computers could use appropriate machine instructions depending on whether arithmetic was being performed with integers or real numbers.

2.2 Dependently Typed Programming

Dependent types [Martin-Löf, 1971] allow the behaviour of programs to be described more accurately than is possible with simple types. The underlying idea of dependent types is that types are allowed to depend on values. The common example used is to define the type $\mathsf{vect}\ \mathsf{A}\ \mathsf{n}$, where A is some type, for representing lists of length n, where each list contains elements from type A. For example, given that nat represents the type of natural numbers, each term belonging to the type $\mathsf{vect}\ \mathsf{nat}\ 2$ represents a list containing two nat items. By making use of the vect type, we can verify program properties concerning list lengths at compile time. We give a definition for vect along with some examples in §2.2.5.

Due to their expressive power, dependently typed languages have a long history of being used as the foundation of theorem proving systems. Such systems include NuPrl [Constable et al., 1986], Coq [Bertot and Castéran, 2004], LEGO [Pollack, 1994] and Alf [Magnusson and Nordström, 1994]. More recently, languages have been designed for the purpose of programming with dependent types. These languages include Cayenne [Augustsson, 1998], Agda [Coquand, 1998], Epigram [McBride and McKinna, 2004] and ATS [Cui et al., 2005].

In this section, we describe the uses of dependent types in programming and introduce the general features of dependently typed languages. As further reading, Altenkirch et al.'s paper on "Why Dependent Types Matter" gives a practical introduction to these topics [Altenkirch et al., 2005].

2.2.1 Uses of Dependent Types in Programming

In the following, we summarise some of the practical applications of dependent types to software development:

Program verification: Dependent types have been used in previous work to capture useful program properties. Examples of this include verifying that arrays are always accessed safely [Xi and Pfenning, 1998], verifying the correctness of a stack-based compiler [McKinna and Wright, 2006] and verifying the properties of finger trees [Sozeau, 2007a].

Expressivity: Dependent types can be used to describe functions that cannot be given satisfactory types in simply typed languages. The common example is to use dependent types to define a C-style printf function, where the type of this function is computed from the input formatting string [Augustsson, 1998]. Moreover, dependent types can be used to reason that certain branches of a program will never be executed, avoiding the need to include redundant error handling code [Xi, 1999a].

Optimisations: When the machine knows more about the static behaviour of a program, additional optimisations can be made at compile time. Dependent types have been used to safely remove dynamic array bounds checks [Xi and Pfenning, 1998], eliminate the checks associated with dead code [Xi, 1999a] and eliminate the need to store run time type tags when implementing type safe language interpreters [Augustsson and Carlsson, 1999].

Outside of the dependently typed programming community, there are of course many other approaches for developing verified software. For example, two of the most well known software verification tools are Spec# [Barnett et al., 2005] and ESC/Java [Leino et al., 2000]. These extend C# and Java respectively with support for describing program specifications using Hoare-style pre and post-conditions. However, dependently typed languages allow for more expressive specifications to be captured than is possible in either of these tools. In particular, specifications in dependently typed languages such as Coq, Epigram and ATS can incorporate any program term (including higher-order functions) which is not possible in Spec# or ESC/Java.

2.2.2 The Curry-Howard Isomorphism

The Curry-Howard isomorphism [Howard, 1980] describes the correspondence of types as propositions and programs as proofs. In other words, given that type T can be viewed as a propositional statement P, a term t that inhabits T is simultaneously a program of type T and a proof that P holds. By using types to describe program specifications, dependently typed languages therefore give an approach to programming and proving in the same language.

2.2.3 Type Erasure

Type erasure is the process of removing information related to types from a program, typically for the purpose of efficiency when the program is run. With simply typed languages, we usually consider the type checking phase of compilation to be separate from the run time program execution phase. This *phase distinction* is not so clear in dependently typed languages which can make type erasure problematic [Mckinna and Brady, 2005]. Coq addresses this issue by providing an explicit type for proofs called **Prop** and a type for computational terms called **Set** (see §2.2.4.1 for more details) so that type erasure becomes a relatively simple process.

2.2.4 A Brief Introduction to Coq

As we make use of Coq for demonstrating our ideas in this thesis, we use Coq as a formal notation to give illustrative examples of dependently typed programming in the following sections. We are only concerned with describing the general concepts of dependently typed programming at the moment but we go into more specific details about programming in Coq in Chapter 2.

The core language used by Coq [Bertot and Castéran, 2004] is based on the Calculus of Inductive Constructions (CIC), a variant of intuitionistic type theory. CIC is both a dependently typed functional programming language and a constructive logic. In other words, we can use CIC to write regular functional programs and verify properties about those programs within the same language. Coq is designed to adhere to the de Bruijn principle [de Bruijn, 1980], where the correctness of a Coq proof relies only on a small trusted kernel.

Notable uses of Coq include the formal proof of the Four Colour Theorem [Gonthier, 2007] and the construction of a verified compiler for a large subset of the C programming language [Leroy, 2009]. Most would not consider Coq as a practical everyday programming language due to the level of experience required to write Coq programs and work with formal proofs. However, recent work, which we cover in §3.3, has aimed to make programming with dependent types in Coq more convenient [Sozeau, 2007b, Sozeau, 2008].

In this section, we give a brief introduction to Coq, starting with its features that have more in common with simply typed languages before discussing concepts of dependently typed languages. We make use of the following Coq conventions:

- t:T means "term t has type T".
- $t_1 \ldots t_n$:T means "the terms $t_1 \ldots t_n$ have type T".
- id := t:T means "id is an identifier with the value t of type T".

2.2.4.1 Sorts

In Coq, the type of a type is referred to as a *sort*. There are two standard sorts called **Prop** and **Set** [Coq development team, 2006]:

- The **Prop** sort is the type of propositions. Given x:P and P:**Prop**, x can be interpreted as a proof that proposition P holds.
- The **Set** sort is the type of program specifications. Given x:S and S:**Set**, x can be interpreted as a program that satisfies the specification S.

When writing Coq programs, we generally want terms that are computationally relevant to have the type **Set** and terms that are only required for reasoning about program behaviour at compile time to have the type **Prop**. Terms of type **Prop** are erased during Coq's program extraction process [Paulin-Mohring, 1989].

Sorts themselves have types. There is a family of infinite sorts called **Type**(i), where i is a natural number. The types of the various sorts are **Prop**:**Type**(0), **Set**:**Type**(0) and **Type**(i):**Type**(i + 1). The type index of a **Type** term is usually left implicit in Coq programs.

2.2.4.2 Inductively Defined Types and Function Definitions

Inductively defined types are a common tool used for representing data in simply typed functional languages. For example, Peano numbers can be defined inductively in Coq with the following ML-like definition:

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

The above definition introduces nat as a new type with the constructor O (note the uppercase letter) to represent zero and S n to represent the successor of n. We can now construct nat terms such as S O and S (S O) to represent the numbers “one” and “two” respectively. 0, 1 and 2 are used as syntactic sugar for the corresponding nat terms.

The following function to sum two nat terms can be defined in the usual functional programming style, using pattern matching and recursion:

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (plus p m)
  end.
```

We use the infix notation + for plus in later examples.

Inductive types can be parametrised by another type to create a family of inductive types. For instance, an inductive type for polymorphic lists, parametrised by a type A, can be defined in Coq as follows:

```
Inductive list (A:Type) : Type :=
  | nil  : list A
  | cons : A → list A → list A.
```

The constructor nil constructs the empty list and, given a head element h and another list t, cons concatenates h onto t. Notice that the use of the **Type** sort allows the list type to be parametrised by types that have type Sort or type **Prop**.

We make use of the notation h::t for cons h t, [] for nil and [x_1; ... ; x_n] as shorthand for x_1 :: ... :: x_n :: nil. Note that, for brevity, we sometimes leave type parameters implicit in examples.

As further examples and to introduce common function definitions, the following shows the length function for calculating the length of a list and the app function for appending two lists:

```
Fixpoint length (A:Type) (a:list A) : nat :=
  match a with
  | [] ⇒ O
  | h :: t ⇒ S (length t)
  end.

Fixpoint app (A:Type) (a b:list A) : list A :=
  match a with
  | [] ⇒ b
  | h :: t ⇒ h :: app t b
  end.
```

We make use of the infix notation of ++ in later examples for app.

2.2.5 Dependent Types

We use the phrase *full dependent types* when dependent types can depend on any value. As we will discuss in §2.3, some languages, like DML, make use of only restricted forms of dependent types. For now, we discuss the general features common to languages with full dependent types like Agda, Epigram, ATS and Coq.

2.2.5.1 Dependent Function Types

Dependent function types generalise the usual function space by allowing functions to be defined that have the type ∀ (x:A), B, where A and B are types and B can depend on x (i.e. second-order polymorphism is allowed). Dependent function types are also known as Π types and dependent product types. In the special case when x does not occur free in B, we simply write A → B.

Recall that we use the type vect A n to represents lists of length n. We can use dependent function types to accurately describe the length of the list returned by a function vapp for appending two vect terms. A suitable type for such a function is as follows:

```
vapp : ∀ (A:Set) (n m:nat),
  vect A n → vect A m → vect A (n + m).
```

We give definitions for vect and vapp over the next two sections.

2.2.5.2 Inductive Families

Dependently typed languages differ from simply typed languages in that they allow inductive types to be defined that are parametrised by a value. Such types are commonly referred to as *inductive families* [Dybjer, 1991]. The value that a type is parametrised over is sometimes referred to as the *type index*.

As mentioned previously, the traditional example of an inductive family is the type vect n, that represents lists of length n. The vect type is defined by indexing a list data structure with a natural number representing the list length, such as in the following definition:

```
Inductive vect (A:Set) : nat → Set :=
  | vnil : vect A O
  | vcons : ∀ (n:nat), A → vect A n → vect A (S n).
```

The vnil constructor creates the empty list with type index O, representing a list of zero length. The vcons constructor concatenates an item onto a list of length n, creating a list of the expected size of length S n. Notice that the constructors have dependent function types.

2.2.5.3 Capturing Program Properties

The type indices of inductive families can be used to statically verify properties of data structures. As an illustrative example of this, we now define a Coq function for appending vect lists:

```
Fixpoint vapp (A:Set) (n m : nat)
  (a : vect A n) (b : vect A m) : vect A (n + m) :=
  match a with
  | vnil ⇒ b
  | vcons n' h t ⇒ vcons h (vapp2  t b)
  end.
```

The output of vapp states that the resultant list should have the same length as the sum of the lengths of the input lists. The function takes five parameters: a type parameter A, the length of the two input lists (parameters n and m) and the two vect list terms to append (parameters a and b). The form of this definition is similar to app from §2.2.4.2, with the exception of the added type indices.

Notice that Coq automatically infers that the type parameter for vcons and vapp in the body of the function must be A. Additionally, Coq is able to infer the value of the type indices for these constructors. However, Coq requires that the type indices are given in each pattern matching expression i.e. we must write vcons n' h t instead of vcons h t in the step case pattern matching expression.

2.2.5.4 Type Equality

Coq's type checker will verify that vapp is well typed and thus always returns a list of the expected length. To help describe what type checking dependently typed programs involves, we first consider the type of the term required for each pattern matching clause and how this compares to the type of the term given:

- For the vnil case, the type expected is vect (0 + m). The term given for this case is b, which has type vect m.
- For the vcons case, the type expected is vect (S n' + m). The term given for this case is vcons h (vapp t b), which has type vect (S (n' + m)) as vapp t b has type vect (n' + m).

In both cases, the indices of the expected type and the actual type differ. The definition vapp type-checks because Coq uses *intensional* type equality. Intensional equality dictates that two types are equal if the types have the same *normal form*.

We explain how normal forms are calculated in Coq in §3.1.1. For now, it suffices to say that this involves applying standard reduction rules to terms until no more reductions apply. We now reconsider the cases from the vapp example as follows:

- For the vnil case, the normal form of the expected type vect (0 + m) is vect m, which matches the type of term b.
- For the vcons case, the normal form of the expected type vect (S n' + m) is vect (S (n' + m)), which matches the type of term vcons h (vapp t b).

In Coq, only total functions are allowed and calculating the normal form of a type is a decidable operation (see §3.1.1).

Another approach when comparing types is to use *extensional* type equality. This notion of equality says that two types are equal if both types are inhabited by the same members. Determining this typically requires constructing proofs as part of type checking. This notion of equality therefore generally leads to undecidable type-checking. For instance, NuPrl [Allen et al., 2000] uses extensional equality when type checking dependently typed functions and has undecidable type checking. Extensional equality tends to be more generous about what types are equal compared to intensional equality.

2.2.5.5 Proof Construction

When the unrestricted use of dependent types is allowed, proofs must typically be constructed when writing programs. We now give an example of a program where proofs are required to be written.

The following attempt to implement list reversal for vect terms looks as if it should be accepted by Coq but is actually ill-typed (the need for the **in** ... **return** annotation is not important to this example):

```
Fixpoint vrev (A:Set) (n:nat) (a:vect A n) : vect A n :=
  match a in vect _ n return vect _ n with
  | vnil ⇒ vnil A
  | vcons n' h t ⇒ vapp (vrev t) (vcons h (vnil A))
  end.
```

Coq's type checker will report that this program is ill-typed with the following message pinpointing the vcons case as the problem:

> The term "vapp (vrev A n' t) (vcons h (vnil A))" has type "vect A (n' + 1)" while it is expected to have type "vect A (S n')".

This error arises because vect A (n' + 1) is not the same type as vect A (S n') under intensional type equality as the type indices do not share the same normal forms. The + function is defined recursively on the first argument, meaning that computations with + involve examining the structure of the first argument. As n' + 1 has a variable in the recursive position, it therefore will not reduce to and match S n.

One approach to fixing the faulty vrev function is to justify that n' + 1 means the same as S n' with a proof of the lemma plus_S : ∀ n, n + 1 = S n. This can be achieved here with the help of eq_rec which is a standard Coq term that can be used to substitute some term x with some term y given a proof of x = y. In other words, eq_rec is a theorem that represents the substitution property of equality.

To fix the error message given for vrev, we can make explicit use of eq_rec with plus_S to justify that the type index for the type of vapp (vrev t) (vcons h vnil) has the same meaning as the expected type index. The following script makes use of this approach to give a well-typed version of the vrev function (where the terms that go in place of the _ symbols are implicit and can be determined by Coq):

```
Fixpoint vrev (n:nat) (a:vect n) : vect n :=
  match a in vect n return vect n with
  | vnil ⇒ vnil
  | vcons n' h t ⇒
    eq_rec _ _
      (vapp (vrev t) (vcons h vnil)) _ (plus_S n')
  end.
```

When we refer to providing proof automation for dependently typed programming, we are referring to machine support for constructing proofs such as the one needed in the above example.

2.2.5.6 Dependent Sums

Dependent sum types, also known as Σ types, represent pairs where the type of the second component can depend on the first. Typically, the first component is a value and the second component is a proof that some property holds for this value.

Coq includes definitions for what are called weak and strong dependent sums. Strong dependent sums corresponds most closely to the standard notion of dependent sums. When performing pattern matching on a strong dependent sum term, both components are accessible, whereas for weak dependent sums only the first component is accessible. We favour the terminology of *subset types* in this thesis to refer to weak dependent sums.

Given A : **Type** and P : A→**Prop**, a Coq subset type is denoted as {x:A | P}. A member of this subset type is constructed using a term y : A and a term of type P y. As an example of a subset type, the type {x: list A | length x = n} describes lists of length n.

This particular subset type offers an alternative representation to the type vect A n for representing a list of a fixed length. Notably, the subset type representation makes use of the type list to represent the list of items and the simply typed length function is used to describe the number of items. In comparison, the vect inductive family represents the list structure by itself and the type indices of the inductive family are used to describe the number of items in the list.

Related to the above, the notation {A}+{B} is used in Coq to denote the constructive sum of the propositions A and B. This is represented as the type sumbool using the following definition:

```
Inductive sumbool (A B:Prop) : Set :=
  | left  : A → sumbool A B
  | right : B → sumbool A B
```

Constructive sums can be thought of as a more informative version of booleans. For example, a function that decides whether two nat terms are equal could have the following type:

```
nat_eq_dec: ∀ (x y:nat), {x = y}+{x ≠ y}
```

The nat_eq_dec x y function returns either a proof of x = y or a proof of x ≠ y.

2.2.6 Recursion and Termination

Most popular programming languages allow unrestricted recursion and it is not uncommon for programming errors to result in programs that fail to terminate at runtime. As type checking dependently typed languages can involve performing computations at compile time, non-terminating functions can cause type checking to loop. For this reason, languages based on type theory, like Coq, typically only allow terminating functions to be defined.

A common restricted form of recursion that ensures termination is *structural recursion*, where the argument to the recursive call must be structurally smaller than the argument to the parent call. For example, vapp (see §2.2.5.3) and app (see §2.2.4.2) were defined by structural recursion. To define a structurally recursive function in Coq, we can make use of the **Fixpoint** command using the following syntax:

```
Fixpoint f (x₁:A₁) ... (xₙ:Aₙ) {struct xᵢ} : T := t
```

This introduces the function definition f that has the parameters $(x_1{:}A_1)\ldots(x_n{:}A_n)$ and returns a term of type T. The function f is implemented by the term t. The expression {**struct** x_i}, where x_i is one of the function parameters, specifies which argument becomes smaller at each recursive call. As an example of using the **Fixpoint** command, the following code will introduce a function called plus, where plus is defined by structural recursion on the first argument:

```
Fixpoint plus (n:nat) (m:nat) {struct n} : nat :=
match n with
| O ⇒ m
| S p ⇒ S (plus p m)
end.
```

Coq can usually infer the structurally recursive parameter and the **struct** expression can be left out in many cases.

Sometimes we may wish to define functions that are not structurally recursive. One approach to defining such functions in type theory is to make use of an accessibility predicate to prove that recursion is well-founded [Aczel, 1977]. In §3.3.4, we give an example of how to define a non-structurally recursive function in Coq.

2.2.7 Impossible Cases

Dependent types can be utilised to prove that certain program branches will never be executed at run time. We call such branches *impossible cases*. For example, consider implementing a function hd that returns the head of a list. The programmer must decide what to do when the function is applied to the empty list. In simply typed languages, choices here include returning a default value, returning an option type, throwing an exception or defining hd as a partial function.

We can use dependent types to specify that hd can only be applied to non-empty lists by giving an appropriate type to hd, such as the following:

```
hd : ∀ (A:Type), {a:list A | a ≠ []} → A
```

Here, we make use of a subset type to specify that the input list to hd cannot be empty. The hd function can be defined as usual by pattern matching on the input list, but for the case of the empty list, we can use the assumption that the input list must be non-empty to prove that such a scenario is impossible. We give a definition for hd in §3.3.3.

2.3 Dependently Typed Programming Languages

We now give an overview of the dependently typed programming languages that are currently available. As the line between proving and programming is blurred with dependent types, it can become unclear at times where to draw a distinction between proof assistants and programming languages. Our intention is to survey systems that are primarily intended to be used for programming with dependently types, as opposed to proof development.

We begin with languages, such as DML [Xi, 1998] and Concoqtion [Fogarty and Pasalic, 2007], where dependent types have been added to existing simply typed languages with varying levels of restrictions. We then discuss languages with full dependent types based on type theory, such as Epigram [McBride and McKinna, 2004] and Agda [Norell, 2007, Bove et al., 2009].

2.3.1 DML

DML [Xi, 1998] is a conservative extension of the ML programming language that allows the use of a restricted form of dependent types. Type indices are restricted to what can be represented in a constraint domain C, where type checking involves solving constraint satisfaction problems in C. The authors demonstrate DML with C instantiated to the domain of linear integer arithmetic. Type checking in this domain is decidable and a variant of Fourier-Motzkin method [Dantzig and Eaves, 1973] is used in DML's implementation for constraint solving. Examples of the program properties that can be captured in DML include verifying that array accesses are safe and that binary tree operations preserve tree balancing properties.

2.3.2 ATS

The obvious limitation of the current implementation of DML is that type indices are limited to the domain of linear arithmetic. The ATS language extends DML with an expressive type system for encoding arbitrary program properties [Cui et al., 2005]. Automation for linear arithmetic proofs is still provided but the user is expected to construct any other required proofs themselves in the form of total functions. ATS uses a Coq-like separation for the logical and computational parts of programs, where the former is insulated from problematic features, such as side-effects and general recursion, used in the language for computations.

2.3.3 Sage

Sage [Gronski et al., 2006] is a pure functional programming language with dependent types where any program term can appear in a type. Proof obligations, generated during type checking, are translated into a form that can be processed by an external theorem prover. The current implementation makes use of the Simplify theorem prover [Detlefs et al., 2005] for type checking, which is assumed to cope well with linear arithmetic theorems. There appear to be no facilities in Sage for the user to write proofs when this automation fails.

An interesting feature of Sage is that program properties that cannot be verified or refuted statically are enforced at run time using a dynamic check. This idea presents a practical solution for when we wish to experiment with a dependently typed program and are unable or unwilling to construct the proofs necessary for type checking. If a runtime check in a Sage program fails during execution, the counterexample found is added to a database. This database is used at compile time to statically detect future errors of the same form.

2.3.4 Concoqtion

Concoqtion [Fogarty and Pasalic, 2007] refers to an approach for extending the type system of an existing language by using a constructive type theory, where the latter has existing proof checking software available. The logical and computational languages are kept separate to allow for decidable type checking. The Concoqtion approach is demonstrated with MetaOCaml Concoqtion, which conservatively extends MetaOCaml [Calcagno et al., 2003] with indexed types. Coq terms are used as the logical language for type indices, where type checking involves the use of the Coq theorem prover. Programs scripts can include Coq proofs and make use of Coq's tactics and decision procedures.

2.3.5 Cayenne

Cayenne [Augustsson, 1998] is a Haskell-like language where type indices can depend on any program expression. As Cayenne programs can make use of unguarded general recursion, it is possible for a non-terminating term to appear in a type. As type checking involves term evaluation, the type checker can thus can fail to terminate and typechecking is undecidable. As a workaround, the number of reduction steps performed while type checking a term can be given an upper bound. The type checker is then sometimes unable to tell whether a term is well-typed or not. To allow type erasure, there are no constructs in Cayenne that allow programs to depend on a type.

2.3.6 Epigram

Epigram [McBride and McKinna, 2004] is a functional programming language with full dependent types. Epigram programs are elaborated to an intensional type theory based on the UTT [Luo, 1994], which is a strongly normalising language with decidable type checking. There is no explicit separation between the logical and computational language in Epigram programs.

Epigram programs are written in a structured editor that is similar in style to the editors for Alf [Magnusson and Nordström, 1994] and Agda [Coquand, 1998]. Programs in Epigram are constructed incrementally by invoking tactics to construct terms. These terms can contain holes that represent missing subterms that the user must complete. Epigram however includes little in the way of proof automation and even simple arithmetic proofs must be proven by hand.

2.3.7 Agda

Agda [Norell, 2007, Bove et al., 2009] is a dependently typed programming language based on intuitionistic type theory which has many similarities to Epigram. Unlike Coq, Agda does not support tactic based theorem proving and instead relies on terms being manipulated by hand. Moreover, there is no Coq-like distinction between computational and logical terms in Agda.

Programming in Agda has been made practical with the introduction of a program testing tool [Dybjer et al., 2003b, Dybjer et al., 2003a, Qiao Haiyan, 2003] and a tool for automating inductive proofs [Lindblad and Benke, 2006]. However, as far as we know, these tools have not been integrated in any way.

2.3.8 Idris

Idris [Brady, 2008] is a language with full dependent types that is described as being closely related to Epigram and Agda. Idris has Haskell-like syntax and is implemented on top of a theorem prover for Haskell called Ivor [Brady, 2007]. A compelling practical feature of Idris is that it supports I/O operations and communication with external programs. Similarly to Epigram and ATS, Idris does not offer any significant proof automation to make dealing with proof obligations easier.

2.4 Inductive Theorem Proving

We now move onto discussing aspects of automated theorem proving. Specifically, *inductive theorem proving* is an important technique for reasoning about functional programs and is the focus of much of our proof automation work.

Induction is a common technique for reasoning about recursively defined data structures and recursively defined functions. Induction is usually performed on a free variable in the goal using a suitable induction principle. For example, when declaring a new inductive data type, Coq will automatically generate a standard induction principle for that type. The following induction principle is automatically generated for Coq's standard list type:

```
∀ (A:Type) (P:list A → Prop),
  P [] →
  (∀ (h:A) (t:list A), P t → P (h :: t)) →
  (∀ (x:list A), P x)
```

As an example of an inductive proof, consider the following theorem:

```
∀ (A:Type) (a b:list A), length (a ++ b) = length a + length b
```

To prove this theorem, we can proceed by induction on variable a, using the standard induction principle for the list type given above. We must then prove two subgoals. The first subgoal is the base case goal of the inductive proof is as follows, which is trivially true by reflexivity in this example:

```
length ([] ++ b) = length [] + length b
```

The second goal is the step case goal of the inductive proof which is as follows:

```
IH : ∀ b, length (a ++ b) = length a + length b
________________________________________________________
length ((h :: a) ++ b) = length (h :: a) + length b
```

The assumption introduced in the step case is usually referred to as the *inductive hypothesis*. Proving the step case of an inductive proof generally relies on transforming the conclusion of the goal so that the inductive hypothesis can be utilised.

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 [Bundy, 2001]. Moreover, proof search includes choices such as the following:

Variable choice: When a conjecture includes several variables, we must decide which variable or variables to perform induction on.

Induction principle choice: Sometimes the standard induction principle is not appropriate for the proof at hand. We may have to choose from a selection of induction principles and sometimes we may even need to invent a new one.

Generalisation: Instead of performing induction directly on the goal we are trying to prove, it is sometimes necessary to prove a more general version of the original goal first. We discuss this technique more in §2.8.

2.5 Rippling

Rippling [Bundy et al., 2005] is an automated theorem proving technique that has been successfully used to automate nontrivial inductive proofs. This technique has been implemented in several theorem provers, including Clam [Bundy et al., 1990], INKA [Hutter and Sengler, 1996], NuPrl [Pientka and Kreitz, 1998b] and Isabelle [Dixon and Fleuriot, 2003].

The use of rippling is a central feature of our proof automation. We give a brief overview of the rippling approach in this section. A more formal and in-depth discussion of rippling can be found in [Bundy et al., 2005].

2.5.1 Overview

Rippling applies whenever a theorem (labelled the *given*), shares syntactic similarities with the conclusion (labelled the *goal*). Rippling directs a proof attempt by using rules that reduce syntactic differences between the goal and the given, where the aim is to utilise the given to advance the proof.

When applied to inductive theorem proving, the inductive hypothesis in the step case of an inductive proof is considered to be the given. Rippling is applicable here as, with typical induction principles, the inductive hypothesis and the goal share syntactic similarities. Application of the inductive hypothesis to the conclusion is usually crucial to proving the step case.

In traditional rippling proofs, modification of the goal is only allowed if differences are reduced with respect to the given. This requirement greatly restricts which rules can be applied. Difference reducing rules are called *wave rules*. As differences can only be reduced a finite number of times in proofs, rippling always terminates.

The rippling technique is known to provide advantages over typical simplification tactics. For example, rippling can guide the use of associativity and commutativity lemmas in step case proofs in ways that will allow the inductive hypothesis to be applied [Bundy et al., 2005]. Lemmas such as these can lead to non-terminating behaviour when used naively as part of simplification. Rippling can also guide the use of case splits in proofs where, in comparison, the naive use of case splitting during simplification can lead to non-terminating behaviour [Johansson, 2009].

2.5.2 Differences and Embeddings

Consider again the step case of the inductive proof described in §2.4 from a rippling perspective. The given and the goal are as follows:

$$\begin{array}{rrcl} \text{Given:} & \forall\, \mathsf{b},\ \mathsf{length}\ (\mathsf{a} \mathbin{+\!+} \mathsf{b}) & = & \mathsf{length}\ \mathsf{a} + \mathsf{length}\ \mathsf{b} \\ \text{Goal:} & \mathsf{length}\ ((\mathsf{h} :: \mathsf{a}) \mathbin{+\!+} \mathsf{b}) & = & \mathsf{length}\ (\mathsf{h} :: \mathsf{a}) + \mathsf{length}\ \mathsf{b} \end{array}$$

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

$$\mathsf{length}\ ((\boxed{\mathsf{h} :: \boxed{\mathsf{a}}}) \mathbin{+\!+} \mathsf{b}) = \mathsf{length}\ (\boxed{\mathsf{h} :: \boxed{\mathsf{a}}}) + \mathsf{length}\ \mathsf{b}$$

Intuitively, we know that the differences have been correctly annotated when removing the annotated terms produces the given. As such, the term that was used as the given can be inferred from the annotated term.

Each collection of shaded terms that represent a difference is referred to as a *wave front*. The unshaded subterm within each wave front, which is part of the given, is referred to as a *wave hole*. Note that a goal can have multiple valid annotations. When we can annotate a goal in the above way, we say that an *embedding* exists and that the given *embeds* into the goal.

2.5.3 Fertilisation

When the entire given matches a subterm within the goal, the matching term in the goal can be replaced with True. This step is called *strong fertilisation*.

When the given is an equation and one side of this equation matches a term t in the goal, the given can be used as a rewrite rule to replace t. This step is called *weak fertilisation*.

Rippling proofs usually involve "rippling-out" wave fronts to the top of the term tree of the goal to allow fertilisation to occur. In rippling proofs, we indicate the direction in which a wave front is moving during the proof attempt with a small arrow on the right of the wave front. Wave fronts that are being rippled-out are indicated with an upwards arrow.

Differences can also be "rippled-in", shown with a downwards arrow on the right of the wave front, to allow fertilisation. When the given contains a universally quantified variable x, strong or weak fertilisation can still occur if differences are moved next to the position in the goal that corresponds to the position of x. Any differences moved to this position can be used to instantiate x in the given when fertilising. The position of terms like x in the given are referred to as *sinks*. The b variable in the annotated term from the previous section is a sink variable. We can indicate b is in a sink position by annotating the variable as follows: $\lfloor \mathsf{b} \rfloor$.

2.5.4 Ripple Measures

Rippling proofs make use of a metric to determine if a transformation has reduced the differences between the goal and the given. For example, to calculate the "sum of distances" measure for an annotated goal, we sum the distance from each outward wave front to the top of the term tree and add this to the sum of the distance from each inward wave front to its closest sink [Dixon and Fleuriot, 2004]. The measure of the goal can therefore be reduced by moving outward wave fronts towards the top of the term tree and moving inward wave fronts towards sinks. Figure 2.1 gives a concrete example of how to calculate the measure of an annotated term.

2.5.5 Example: A Rippling Proof

Figure 2.2 shows a rippling proof of the step case example from §2.5.2. This proof makes use of the following lemmas (note that it would be up to the user to decide on the set of lemmas that should be used during rippling proofs):

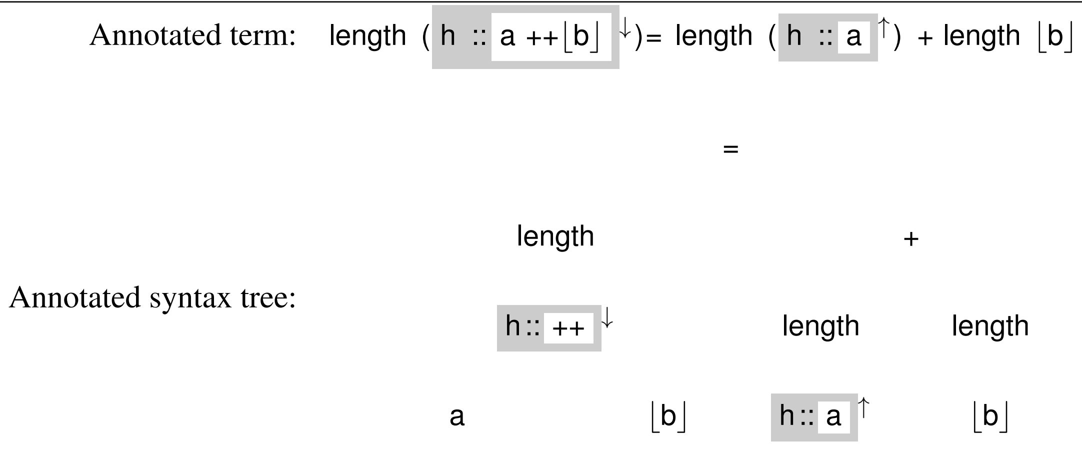


Figure 2.1: The above shows the syntax tree for an annotated term. Nodes are decorated with wave fronts that indicate the differences in the goal. The inward wavefront is 1 step from its nearest sink and the outward wavefront is 3 steps from the top of the term tree. Thus, the measure for this annotation is 1 + 3 = 4.

$$\begin{array}{lrcl}
w1: & \forall\, h\ t\ b,\ (h :: t) \mathbin{++} b & = & h :: (t \mathbin{++} b) \\
w2: & \forall\, h\ t,\ \text{length}\ (h :: t) & = & S\ (\text{length}\ t) \\
w3: & \forall\, x\ y,\ S\,x + y & = & S\ (x + y) \\
w4: & \forall\, x\ y,\ x = y \rightarrow\ S\,x & = & S\,y
\end{array}$$

2.6 Proof Planning and Critics

Proof planning is an approach that provides high-level guidance to a proof attempt with the use of so-called proof plans [Bundy, 1988]. A proof plan can be thought of a high-level outline of the proof that we intend to generate. The purpose of a proof planner is to construct an appropriate proof plan for the goal we wish to prove, where this proof plan will then be used to guide to proof search.

Proof plans are composed of methods. A method is composed of a tactic along with a formal specification of the pre-conditions and post-conditions for applying that tactic. For example, given a rippling tactic, a rippling method that uses this tactic could have the pre-condition that the goal must contain an assumption that embeds into the conclusion of the goal and the post-condition could be that strong fertilisation must have taken place when the tactic succeeds.

A *critic* describes a strategy for fixing failed proof attempts during the execution

$$\text{length } ((\boxed{h :: \boxed{a}}^{\uparrow}) ++\lfloor b \rfloor) \quad = \quad \text{length } (\boxed{h :: \boxed{a}}^{\uparrow}) + \text{length } \lfloor b \rfloor$$

$\Downarrow$ LHS rippled out using w1.

$$\text{length } (\boxed{h :: \boxed{a ++\lfloor b \rfloor}}^{\uparrow}) \quad = \quad \text{length } (\boxed{h :: \boxed{a}}^{\uparrow}) + \text{length } \lfloor b \rfloor$$

$\Downarrow$ LHS rippled out using w2.

$$\boxed{S\ \boxed{(\text{length } (a ++\lfloor b \rfloor))}}^{\uparrow} \quad = \quad \text{length } (\boxed{h :: \boxed{a}}^{\uparrow}) + \text{length } \lfloor b \rfloor$$

$\Downarrow$ RHS rippled out using w2.

$$\boxed{S\ \boxed{(\text{length } (a ++\lfloor b \rfloor))}}^{\uparrow} \quad = \quad \boxed{S\ \boxed{(\text{length } a)}}^{\uparrow} + \text{length } \lfloor b \rfloor$$

$\Downarrow$ RHS rippled out using w3.

$$\boxed{S\ \boxed{(\text{length } (a ++\lfloor b \rfloor))}}^{\uparrow} \quad = \quad \boxed{S\ \boxed{(\text{length } a + \text{length}\lfloor b \rfloor)}}^{\uparrow}$$

$\Downarrow$ Final differences removed using w4.

$$\text{length } (a ++\lfloor b \rfloor) \quad = \quad \text{length } a + \text{length}\lfloor b \rfloor$$

Figure 2.2: An example rippling proof for the step case from §2.5.2. After rewriting the conclusion with a lemma, we recalculate the differences between the modified conclusion and the given. Notice that each step reduces the differences between the goal and the given until, at the end, strong fertilisation is possible.

of a proof plan [Ireland and Bundy, 1996]. Critics are attached to methods and are invoked when a method fails in a specific way. Several critics have been developed that are based specifically on patching failed rippling proofs based on different failure conditions.

The most commonly applicable critic is *lemma calculation* which can be used to discover lemmas rippling needs to succeed. Lemma calculation is invoked when there are no more difference reducing rules to apply in the proof attempt and only weak fertilisation is possible. Lemma calculation proceeds by starting a proof of a new conjecture whose form is derived from a weak fertilised and generalised version of the current goal. If this new conjecture is proven, the lemma proven is used to unblock the rippling proof.

2.7 Lemma Discovery

The lemma calculation critic for rippling attempts to conjecture a missing lemma at the stage in a proof where the lemma is needed. An alternative to this lazy approach to lemma discovery is to attempt to eagerly discover lemmas that may be useful prior to attempting proofs. We briefly discuss this domain of lemma discovery for inductive theorems here.

IsaCoSy is a system for Isabelle for discovering inductive theorems [Johansson, 2009]. Given a set of initial theorems, IsaCoSy will generate a set of constraints for how these theorems should be used to generate conjectures. Constraints are generated in such a way as to avoid naively generating conjectures that trivially follow from currently known theorems. A counterexample checker is first used to filter nontheorems from this list of conjectures. IsaPlanner's rippling-based proof automation then attempts to find a proof for each conjecture.

MATHsAiD [McCaslan et al., 2007] aims to discover inductive theorems and employs heuristics to identify theorems that mathematicians would consider interesting, as opposed to generating an exhaustive list of all theorems that can be found. Whereas IsaCoSy generates conjectures to prove, MATHsAiD uses forward-chaining to discover new theorems.

2.8 Generalisation

Generalisation [Aubin, 1976, Boyer and Moore, 1979, Aderhold, 2007] 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. Somewhat counter-intuitively, g' is usually easier to prove than the more specialised goal g. In some cases, generalisation is required before performing induction so that the inductive hypothesis is strong enough for a proof to be found. Following the terminology in [Walther, 1994], some common generalisation approaches are as follows:

Common subterm generalisation involves identifying a set of common non-variable subterms in the conclusion and replacing these subterms by a fresh variable. For example, the statement $(x + 1) + y = y + (x + 1)$ can be made more general by generalising the common subterm $x + 1$ to z to produce the statement $z + y = y + z$.

Generalising apart is performed by replacing only some of the occurrences of a repeating variable in a statement with a fresh variable. For example, the statement $x + x + y = x + y + x$ can be made more general by generalising apart the occurrences of x to z to produce $z + x + y = z + y + x$.

Inverse functionality is used to generalise equations where the top level function used on both sides of the equation match. Inverse functionality is used to generalise statements of the form $f\ x_1\ \ldots\ x_n = f\ y_1\ \ldots\ y_n$ by removing the application of f to produce the statement $(x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)$. For example, $S\ (x + 1) = S\ (S\ x)$ can be generalised to $x + 1 = S\ x$.

Inverse weakening involves removing unnecessary conditions from a statement. For example, the statement $\forall\ x\ y,\ x \neq y \to x + y = y + x$ can be generalised to the statement $\forall\ x\ y,\ x + y = y + x$ as the condition $x \neq y$ is not required to write the proof.

However, generalisation can be an unsafe proof step in that a provable goal can be *overgeneralised* to form a new goal that is not provable. The typical approach is to make use of a counterexample finder to detect overgeneralisations. We discuss counterexample generation in the next section.

2.9 Counterexample Generation

Testing can be used for checking the correctness of a conjecture before attempting a proof and as a light-weight alternative to formal verification. Testing usually refers to checking whether a conjecture holds for some finite number of example instantiations to increase confidence that the conjecture is actually a theorem.

A counterexample to a conjecture is some example that falsifies that conjecture. For a universally quantified conjecture of the form ∀ x, P x, it suffices to find one instance of x such that P x leads to a contradiction to falsify that conjecture. We examine some practical tools for finding counterexamples in this section.

2.9.1 QuickCheck

QuickCheck [Claessen and Hughes, 2000] is a well-known tool for Haskell that offers automated assistance for program testing. The programmer supplies QuickCheck with universally quantified conjectures about the behaviour of their program and QuickCheck tests these conjectures by searching for counterexamples. To check for a counterexample, QuickCheck replaces the universally quantified variables in the conjecture with appropriately typed randomly generated terms. QuickCheck then evaluates the truth of the conjecture with these concrete values. If the conjecture evaluates to false, a counterexample has been found.

2.9.1.1 Generators

Testing statements that include pre-conditions can be challenging as randomly generated data is unlikely to satisfy the necessary conditions. For example, consider testing the following statement where sorted x is true when x is a sorted list and, under the condition that y is a sorted list, insert i y inserts the item i into sorted position into list y:

```
∀ x i, sorted x → sorted (insert i x)
```

Except for small terms, random instantiations of x are highly unlikely to satisfy the condition of being sorted. Naive term generation will result in most test cases being trivially true, leading to poor test coverage.

One approach to this problem in QuickCheck is to use a custom *generator function*. The purpose of a generator is to generate random terms that always satisfy a certain condition. For example, a generator for the condition above would randomly generate sorted lists.

2.9.2 SmallCheck and Lazy SmallCheck

SmallCheck and Lazy SmallCheck offer an alternative approach to QuickCheck's random generation of terms for finding counterexamples [Runciman et al., 2008]. SmallCheck searches for counterexamples by exhaustively testing, up to some fixed maximum term size, all possible term instantiations. This approach has the advantage that the smallest, and thus usually easiest to inspect, counterexample will be found and the need for writing custom generators is reduced.

Lazy SmallCheck makes use of partially-defined inputs to prune large areas of the test space [Runciman et al., 2008]. For example, given that `sorted [2; 1; x]` evaluates to false without evaluating `x`, testing variants of `x` is unnecessary. Lazy SmallCheck results in significant performance improvements in many cases and allows a greater search depth to be tested in less time than with SmallCheck [Runciman et al., 2008].

2.9.3 Testing in Agda

Of particular relevance, a QuickCheck-like tool is available for Agda and has been used in case studies to develop dependently typed programs [Dybjer et al., 2003b, Dybjer et al., 2003a, Qiao Haiyan, 2003]. This tool has been mostly implemented within Agda itself, where custom generators are written as Agda functions. An interesting practical utility of this is that we can formally verify within Agda that such generators have the expected property of being surjective functions. In other words, we can verify that the generator for a type is able to generate all possible terms of that type.

2.10 Summary

We have introduced the primary features of dependently typed programming languages and, in particular, described the need for proof construction when writing programs. We then surveyed the current development environments available for dependently typed programming followed by an introduction to topics concerning automated inductive theorem proving and program testing. In the next chapters, we elaborate further on how dependently typed programs are constructed before introducing our framework for supporting this process.

Chapter 3

Programming with Dependent Types

This chapter describes ways in which dependently typed programs can be constructed and summarises design choices that need to be considered when capturing program properties with types. These topics will become important later when we discuss the design and scope of our framework for supporting dependently typed programming. As we want to provide support for programming with full dependent types, we focus on features common to languages like Agda, Epigram, ATS and Coq. Moreover, we introduce the Russell language for Coq [Sozeau, 2008], which we utilise in our framework prototype.

3.1 Coq

In this section, we describe some specific details related to programming in Coq to provide more clarity to the examples we give later. We recommend [Bertot and Castéran, 2004, Giménez and Castéran, 2005] as further material for learning how to program in Coq and the use of Coq's manual [Coq development team, 2006] as a reference guide.

3.1.1 Reductions, Normalisation and Convertibility

We now describe the reduction rules used in Coq to perform computations. These rules are of particular relevance in their use to compute normal forms when comparing types during type checking (see §2.2.5.4). The notation $\mathsf{t}\{\mathsf{x}/\mathsf{u}\}$ is used to represent the term that results from substituting all free occurrences of x in term t by u, where α-conversion is used to avoid variable capture. The reduction rules used in Coq are as follows [Coq development team, 2006]:

δ-reduction: This reduction is used for unfolding definitions. If id is an identifier with the value v in the current context, then δ-reducing id in the term t results in the term t{id/v}.

β-reduction: A term of the form (fun x ⇒ s) t is called a β-*redex*. Performing β-reduction on such a term results in the term s{x/t}.

ζ-reduction: Performing ζ-reduction on a term of the form **let** x := s **in** t results in the term t{x/s}.

ι-reduction: Informally, this reduction will simplify a pattern matching expression by determining which pattern matches and making the appropriate simplification (see [Coq development team, 2006] for further details).

A term is said to be in *normal form* when none of the above reductions can be applied. In Coq, sequences of reductions on terms have several important properties including:

Strong normalisation: A term can only be reduced a finite number of times and will eventually reach a normal form.

Confluence: If t_1 can be reduced to the terms t_2 and t_3, both t_2 and t_3 can then be reduced to the term t_4.

As reductions are strongly normalising and confluent, all terms have unique normal forms [Bertot and Castéran, 2004]. If two terms can be reduced to the same term by reductions, the terms are said to be *convertible*. For example, the terms 1 + 1 and 2 are convertible because 1 + 1 can be reduced to 2 using a combination of δ, β and ι reduction (these reduction steps were explained at the start of the section). However, if + is defined to be structurally recursive on the first argument, x and x + 0 are not convertible as these terms are already in normal form.

3.1.2 Equality

Equality in Coq is represented as a parametrised inductive definition called eq, which has the following type:

```
eq : ∀ (A:Type), A → A → Prop
```

We write eq A x y as x = y, where the type parameter is implicit. The only constructor for eq is refl_equal, which has the following type:

```
refl_equal : ∀ (A:Type) (x:A), x = x
```

For example, the term 1 = 1 represents a proposition and refl_equal nat 1 is a proof of this proposition. The term 1 = 0 represents a proposition for which there exists no proof. Notice that the definition of = only allows two terms that have convertible types to be compared. This can be problematic when, for example, we want to compare a term of type vect n and a term of type vect (n + 0). McBride's so-called "John Major" equality, which can be defined in Coq, provides an approach for making such comparisons [McBride, 2000].

3.2 Program Construction

In this section, we consider approaches for constructing dependently typed programs. We consider manual term construction and term construction with the use of tactics. We then introduce the Russell language for Coq [Sozeau, 2008] which we utilise in our framework prototype. This language gives a convenient approach to dependently typed programming as it allows the computational and logical parts of a program to be constructed separately.

3.2.1 Constructing Programs Manually

Dependently typed programs can be constructed by working directly with the term language to build programs by hand. The following Coq function for reversing vect lists from §2.2.5.5 was written in this manner:

```
Fixpoint vrev (n:nat) (a:vect n) : vect n :=
  match a in vect n return vect n with
  | vnil ⇒ vnil
  | vcons n' h t ⇒
    eq_rec _ _
      (vapp (vrev t) (vcons h vnil)) _ (plus_S n')
  end.
```

Recall that we were required to add propositional terms to the step case of the function so that the program would be well-typed. Dependently typed functions like the above can be incrementally built by hand using feedback from the type checker to construct terms of the appropriate type.

Proofs in Coq can also be constructed directly in the style of a dependently typed program. However, this approach is uncommon in Coq scripts as terms for relatively simple proofs are large and difficult to interpret. As such, proofs in Coq are usually constructed with machine assistance, which we cover in the next section.

3.2.2 Proof Construction with Tactics

Proof assistants usually include *tactics* that provide machine assistance for incrementally building proofs. In systems based on type theory, tactics are used to build terms that have the same type as the proof of the proposition we want to prove. For example, we can construct a proof for the proposition ∀ n, n + 1 = S n with the following annotated Coq script:

```
Lemma plus_S : ∀ n, n + 1 = S n.
(* Perform induction on n and label the inductive hypothesis H *)
induction n as [|n H].
  (* Base case subgoal: 0 + 1 = 1 *)
  reflexivity. (* Proof by reflexivity *)

  (* In the step case subgoal, the inductive hypothesis
     H is n + 1 = S n and the conclusion is
     S n + 1 = S (S n) *)
  simpl. (* Simplify conclusion to: S(n + 1) = S(S n) *)
  rewrite H. (* Fertilise conclusion to: S(S n) = S(S n) *)
  reflexivity. (* Proof by reflexivity *)
Qed.
```

When the proof begins, the only information known about the structure of the term being built is that the type of the final term should be ∀ n, n + 1 = S n. When the induction tactic is invoked in the script above, the term representing the proof is partially constructed using the induction principle for nat. This partial term contains holes for the subterms that correspond to the base case and step case proof where Coq's user interface shows these holes as subgoals. The next lines in the script discharge these subgoals and instantiate these holes to create the complete term.

3.2.3 Program Construction with Tactics

As well as proofs, tactics can be used to construct dependently typed programs. This is done by specifying the type of the function we want to build and using tactics to incrementally build a term of the corresponding type. For example, the following proof script will build a term with the same type and computational behaviour as the vrev function from §2.2.5.5:

```
Definition vrev : ∀ (A:Set) (n:nat) (a:vect A n), vect A n.
(* Perform induction on term a and name the inductive hypothesis H *)
intros. induction a as [|n' h H].
  (* Base case: vect 0 *)
  exact (vnil A). (* Supply term to use *)

  (* Step case: vect (S n') *)
  rewrite <- plus_S. (* Rewrite conclusion: vect (n' + 1) *)
  exact ((vapp H) (vcons h (vnil A))). (* Supply term to use *)
Qed.
```

Intuitively, when considering the term constructed by this script, the induction step in the proof corresponds to performing pattern matching on the input vect term and the use of the inductive hypothesis (labelled H in the script) corresponds to the recursive call. When writing programs with tactics in this way, it is important to be aware which tactic calls are used to construct computationally relevant terms in the final program.

3.2.4 Constructing Computational and Logical Terms Separately

Sometimes it can be preferable to construct the computational and logical parts of a program separately. For example, this methodology was employed when verifying a Java Card tokenization algorithm in Coq [Denney, 2001, §5]. Coq's *Program tactic* [Sozeau, 2007a, Sozeau, 2007b, Sozeau, 2008] gives a convenient method for constructing programs in this fashion, where we write the computational part of a program first and then write the proof of correctness.

3.3 Dependently Typed Programming in Russell

By using the Program tactic, we can write the computationally relevant parts of a dependently typed function and defer the construction of the required proofs to a later time. Previous work in Coq by Parent [Parent, 1995] provided similar facilities. The Program tactic accepts function definitions written in a language called *Russell* [Sozeau, 2007a,Sozeau, 2007b,Sozeau, 2008]. Russell functions share much of the syntax and typing rules of regular Coq definitions except that Russell permits certain terms to be omitted. After a decidable type-checking procedure, a Russell program is interpreted into a Calculus of Inductive Constructions term that contains uninstantiated typed metavariables in the place of the missing proofs. These missing proofs become *proof obligations* that must be solved to complete the program definition.

The Program tactic will attempt to automatically discharge proof obligations with a configurable tactic. When a proof cannot be found automatically, the user is asked to interactively construct a suitable term using tactics in the form of a Coq proof goal. We give an introduction to programming in Russell in the next few sections. For a more formal treatment of Russell and its typing rules, see [Sozeau, 2007b].

3.3.1 Inductive Family Coercions

When a subterm of a Russell program is expected to belong to an inductive family I with type index x, it is permitted to use a term from the same inductive family I with the type index y as long as we later discharge a proof obligation of the form x = y. We now give an example of a Russell function that generates proof obligations such as this.

Russell definitions are prefixed with the **Program** keyword. The following Russell function uses vect to capture the length of the list returned by a function that concatenates a list of n lists, each of length m, together:

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

As can be seen from the above, Russell programs have the appearance of regular Coq functions. The important difference here is that, for the function body to be a valid Coq program, type coercions would need to be added to each pattern matching clause. As explained at the start of this subsection, Russell allows type coercions to be omitted in certain situations.

Due to the type coercions we omitted, this Russell definition generates a proof obligation for each pattern matching clause. We introduce the terminology *base case proof obligation* for proof obligations produced by the base case term of a function and *recursive call proof obligation* for proof obligations produced by the step case term of a function.

The base case of vconcat generates a proof obligation because the result term vnil A has type index 0 when it is expected to have type index $m * 0$. We therefore need to prove that $\forall$ m, 0 = $m * 0$. The actual proof obligation generated by Program is shown as follows in the form of a Coq proof goal:

```
A : Set
n : nat
m : nat
a : vect (vect A m) n
Heq_n : 0 = n
Heq_a : vnil ≃ a
____________________________
0 = m * 0
```

The proof obligation contains the assumptions A, n, m and a as these are the input terms to vconcat. The assumption Heq_n and Heq_a are generated from the use of pattern matching in the definition of vconcat. The symbol $\simeq$ represents McBride's John Major equality [McBride, 2000] (which we mentioned in §3.1.2).

Briefly, proof obligations contain the information that is deduced about terms when pattern matching is performed [Sozeau, 2007a, §3.3]. In this case, the term a was matched against the pattern vnil so the proof obligation contains the assumption 0 = n and vnil $\simeq$ a. We refer to any equations produced by pattern matching as *pattern matching equations*.

3.3.2 Subset Type Coercions

Russell provides support for programming with subset types (see §2.2.5.6) using a mechanism based on the predicate subtyping feature from PVS [Shankar and Owre, 1999]. The essential idea is that, when a term in a Russell function is expected to have type {x:A | P x}, Russell allows the use of a term t:A if we later provide a proof of P t in the typing context of t.

The proj1_sig t function is a standard Coq definition for returning only the computational part of a subset type term t. If a Russell program includes a term t:{x:A | P} when a term of type A was expected, the Program tactic will use the proj1_sig function to coerce t to the expected type.

We now give an example of a Russell function that includes the use of subset types. The following Russell function reverses the input list a and returns a subset type term that contains the reversed list r with a proof that a and r share the same length:

```
Program Fixpoint srev (A:Set) (a:list A) :
  {r:list A | length r = length a} :=
  match a with
  | []    ⇒ []
  | h::t ⇒ (srev t) ++ [h]
  end.
```

For the base case of the function, we are required to show that the length of the list returned (i.e. []) has the same length as the input list a. This is done by proving the base case proof obligation, which has the following form:

```
A:Set
a:list A
Heq_a:[] = a
____________________________
length [] = length []
```

Again, notice the equational assumption Heq_a that was produced from the use of pattern matching.

The term given for the step case of srev is (srev t) ++ [h]. As ++ expects terms of type list and srev t is a subset type term, the Program tactic uses the proj1_sig function mentioned above to coerce the srev t term to the expected type. For the step case, we therefore must prove (proj1_sig (srev t)) ++ [h] has the expected length. The proof obligation generated for the step case is as follows:

```
srev : ∀ (A:Set) (a:list A), {r:list A | length r = length a}
A : Set
a : list A
h : A
t : list A
Heq_a : h :: t = a
______________________________________________________________
length ((proj1_sig (srev A t)) ++ [h]) = length (h :: t)
```

Proving recursive call proof obligations usually involves making use of the propositional part of the subset type term returned by the recursive call. The first step in doing this is typically to destructure the result term of the recursive call r into its computational part s and propositional part p. This allows terms of the form proj1_sig r to simplify to s and then p can be used as part of the proof. For example, destructuring srev A t and simplifying away the proj1_sig term in the above goal produces the following:

```
srev : ∀ (A:Set) (a:list A), {r:list A | length r = length a}
A : Set
a : list A
h : A
t : list A
Heq_a : h :: t = a
srev_s : list A
srev_p : length srev_s = length t
________________________________________________________________
length (srev_s ++ [h]) = length (h :: t)
```

3.3.3 Impossible Case Proof Obligations

The ! symbol is used in Russell programs to indicate that a particular branch of a program is an impossible case (see §2.2.7). For example, in the following function that returns the head of a list, we mark the base case as being an impossible case:

```
Program Fixpoint hd (A:Type) (a:list A | a ≠ []) : A :=
  match a with
  | nil ⇒ !
  | h::t ⇒ h
  end.
```

Note that, in the above function, (a: list A | a ≠ []) is convenient shorthand notation for a:{x: list A | x ≠ []}.

The use of the ! symbol will produce a proof obligation where we must show that the typing context where the symbol appeared contains a contradiction. For the above program, the generated proof obligation has the form {a: list A | [] ≠ []} → False.

3.3.4 Termination Measures

Only terminating functions can be defined in Coq. Non-structurally recursive functions can be defined using Russell provided we construct the proofs required to show each function terminates. The Russell syntax {measure x} is used to state that a function terminates because, each recursive call, the nat term x becomes smaller each time. For example, we can define a function for calculating Fibonacci numbers using a decreasing measure as follows:

```
Program Fixpoint fib (n:nat) {measure n} : nat :=
match n with
| O ⇒ 1
| S O ⇒ 1
| S (S x) ⇒ fib x + fib (S x)
end.
```

In this specific example, {measure n} states that the value of n passed to each recursive call to fib should always be less than the value of n passed to the parent call. Therefore, to prove this function terminates, we must discharge a proof obligation of the form $\forall$ x, x $<$ S (S x) and another of the form $\forall$ x, S x $<$ S (S x). These proof obligations are produced by the first and second recursive call respectively.

3.4 Program Specifications

When writing dependently typed programs, we want to make use of types in such a way that type checking enforces the program properties we are interested in capturing. In this section, we refer to the type given to a dependently typed function as being its program specification. We now discuss some design choices writing program specifications and describe some of the features of working with different representations.

3.4.1 Strong and Weak Specifications

Specifications can be described as fitting within two general categories:

Strong specifications precisely describe all valid input and output pairs we would expect from a correct function. For example, a strong specification for a list sorting function f is "f returns a sorted list that is the permutation of its input".

Weak specifications only specify some of the behaviour we would expect from a correct function. Such specifications are also referred to as being "loose"or under-specified. For example, a weak specification for the list sorting function f is "f returns a sorted list". This captures some of the properties expected in a sorting function but notice a function that always returns the empty list would satisfy this specification.

3.4.2 Transparent and Opaque Definitions

Proof assistants make use of the notion of *transparent* and *opaque* definitions. Transparent definitions can be unfolded in proofs. Opaque definitions differ in that, by design, they cannot be unfolded and only their typing can be observed.

Dependently typed functions are generally declared opaque to hide their implementation details and only the information given by the typing of such functions can be relied on in proofs. For example, the typing of srev from §3.3.2 only provides a guarantee regarding the length of the list that srev returns. In contrast, simply typed functions are usually transparent when used in the typing or the implementation of a dependently typed function.

3.4.3 Type Refinement Choices

When developing dependently typed programs, we have a design choice of which functions we make dependently typed and opaque, and which functions we make simply typed and transparent. The concept of how specific the typing of a program is at varying levels is known as *type refinement* [Pfenning, 1993]. For example, reconsider the srev function:

```
Program Fixpoint srev (A:Set) (a:list A) :
  {r:list A | length r = length a} :=
  match a with
  | []   ⇒ []
  | h::t ⇒ (srev t) ++ [h]
  end.
```

Here, we have chosen to use a transparent simply typed append function (i.e. ++) in the body of srev. As ++ is transparent, we can write proofs that rely on any of the usual properties of append.

An alternative approach to implementing srev is to replace ++ with an opaque dependently typed append function with the following type:

```
sapp : ∀ (A:Type) (a b:list A),
  {r:list A | length r = length a + length b}
```

If we then implement srev using sapp, a call to sapp will appear in the recursive call proof obligation. Destructuring this call to sapp will produce a propositional term that describes the length of the list sapp returns, resulting in a proof obligation with a different form to before. In several examples in Chapter 9, we explore how varying the level of type refinement in a program can require more challenging proofs.

3.4.4 Functions and Inductive Predicates

Program properties can be described with the use of functions as well as with *inductive predicates*. We describe these representations here.

3.4.4.1 Inductive Predicates

Predicates can be defined using inductive types to create so-called *inductive predicates*. In Coq, the commonly used operators for conjunction ($\wedge$), disjunction ($\vee$) and equality (=) are defined as inductive predicates. For example, the following inductive predicate defines a type that can be used to build a proof that a natural number is even:

```
Inductive p_even : nat → Prop :=
  | even_O : p_even O
  | even_S : ∀ n, p_even n → p_even (S (S n)).
```

The proposition p_even n can be interpreted as "n is even". To prove p_even n, we must show that the type p_even n is inhabited. For example, we can show the proposition p_even (S (S O)) holds by building the witness term even_S even_O. Proofs concerning inductive predicates typically involve determining which constructors should be used to build a term of the appropriate type in this way.

Inversion is a common tool for reasoning about inductive predicates [Cornes and Terrasse, 1995]. Briefly, inversion is used to reason about which constructors could have been used to construct a term of a certain type. For example, given the constructors for p_even, we can reason that p_even 4 must have been constructed with the constructor even_S and a term of type p_even 2. We can also reason that a term of type p_even 1 is impossible to construct.

3.4.4.2 Predicates as Functions

Instead of using inductive predicates, predicates can also be defined as regular functions. For example, the following function f_even returns True when n is even and False otherwise:

```
Fixpoint f_even (n:nat) : Prop :=
  match n with
  | O       ⇒ True
  | S O     ⇒ False
  | S (S p) ⇒ f_even p
  end.
```

In contrast to the inductive predicate p_even from before, we can perform computations with f_even. This time, to check that S (S O) is even, we only have to simplify the term f_even (S (S O)) by computation and check that the result is True. The work in this thesis mainly concerns the use of recursive functions over inductive predicates.

3.5 Conclusions

We have given a summary of the main approaches for constructing dependently typed programs and discussed some of the design choices available when capturing program specifications. We looked at how programs can be built directly by hand and how assistance can be given to term construction with the use of tactics. We then introduced the Russell language that gives a convenient approach to programming in Coq, where tactics can be used to construct any required proofs. The proofs required typically involve manipulating equations and reasoning about inductively defined types. We describe the general pattern of these proofs in Chapter 6.

We then described some of the choices that are available when capturing program specifications, such as the option of which functions in dependently typed programs should be simply typed and how to represent predicates. These choices will become relevant when describing what style of programming we can support with our framework.

Chapter 4

Challenges when Programming with Dependent Types

In this chapter, we discuss some of the challenging aspects of programming with dependent types. Specifically, we explain why we believe user assistance is needed for constructing proofs and coping with errors in programs. The purpose here is to describe the motivation behind the framework we have designed that aims to make dependently typed programming more practical. We introduce this framework in the next chapter.

4.1 User-Defined Properties

In this thesis, we use the phrase *user-defined properties* to refer to program properties, captured with the use of dependent types, that involve data types and functions that were introduced by the user. We aim to support dependently typed programming with user-defined properties and not, for instance, limit the user to only working with some restricted predefined set of definitions. In particular, we want to provide support for capturing program behaviour such as the following with user-defined properties:

Membership properties, such as those involving subcollections and permutations e.g. "reversing a list results in a permutation of the initial list".

Ordering properties, such as stating that a collection is sorted and that an item has been added to particular position in a data structure e.g. "an insertion sort function produces a sorted list".

Program equivalence properties, such as showing that an optimised version of a program produces the same results as an simpler but unoptimised version e.g.

"a tail recursive version of a factorial function gives the same results as a non-tail recursive version".

Arithmetic properties, such as those involving the number of items in a collection and the size/height/depth of a data structure. We are particularly interested in providing some support for non-linear arithmetic properties e.g. "a complete binary tree of depth n has $2^n - 1$ nodes".

We note that typical dependently typed programs will also make use of types and functions that are taken from the standard libraries that form part of the programming environment being used. In this thesis, we primarily focus on providing support for types and definitions introduced by the user.

4.2 Proof Construction

As seen previously, developing dependently typed programs typically requires that we construct proofs (see §3.3). Requiring users to construct proofs themselves makes dependently typed programming less practical for the following reasons:

Proof construction can be difficult. It is well known that constructing formal proofs is a challenging task and can be particularly daunting for beginners. Without suitable proof automation, dependently typed programming will require that users have theorem proving experience.

Proof construction is time consuming. Even when a skilled user is able to construct the required proofs, this process can take a lot of effort. As such, without proof automation, users will be discouraged from capturing properties that involve time consuming proofs.

In this thesis, we focus on capturing user-defined properties that involve inductively defined types and recursively defined functions. As we describe in Chapter 6, we find that capturing these forms of properties typically requires inductive proofs to be written. Inductive proofs can also be commonly seen in Agda, ATS, Coq and Epigram programs. Most users are likely to find writing such proofs manually both challenging and time consuming. As such, we believe proof automation support is important to making programming with user-defined properties more practical.

We note here that there are benefits to constructing proofs manually as, for example, this process can be insightful in understanding the behaviour of a program. However, we leave the discussion of where automation is suitable open to debate.

4.3 Coping with Errors

Recall that in Russell programs, proof obligations had to be discharged to complete the definition of a dependently typed program (see §3.3). When there is a mismatch between the specification of a program, described with the use of dependent types, and the actual behaviour of the program, some of the proof obligations generated will be unprovable. An unprovable proof obligation indicates that there is either an error in the specification of the program, the behaviour of the program or both (see §8.1 for an example of this).

Determining that a proof obligation is unprovable and that there is an error in the program is typically left to the user in most systems. This makes dependently typed programming less practical for the following reasons:

Identifying errors can be challenging. In our experience, errors are difficult to identify by hand and it is not often immediately obvious that a proof obligation is unprovable. In particular, much time can be wasted during development attempting to discharge unprovable proof obligations before an error is noticed.

Fixing errors can be challenging. When we are aware that an error exists, identifying where the fault is and what modifications need to made to correct the problem can be difficult. For example, it can be unclear if there is a fault in the program specification, the program behaviour or both.

We therefore believe support is needed for coping with errors to make dependently typed programming easier.

4.4 Conclusions

We have described aspects of dependently typed programming that we believe need support to make development more practical. We discussed how users are likely to find the need for manual proof construction challenging as this can be both a difficult and time consuming activity. Moreover, we explained why support is needed for coping with errors that are indicated by unprovable proof obligations. In the next chapter, we introduce our framework designed to address these aspects of dependently typed programming.

Chapter 5

A Framework for Supporting Dependently Typed Programming

In the previous chapter, we described how programming with dependent types can be challenging because of the difficulties involved in manually constructing proofs and coping with errors. We now introduce our framework for supporting dependently typed programming that is designed to address these areas. The framework combines ideas from the domains of proof automation and testing, where we include features for automatically discharging proof obligations and giving feedback on errors. In this chapter, we give a high-level description of this framework and how its features are designed to make dependently typed programming more practical.

5.1 Framework Features

We first describe the high-level features provided by our framework. These features are described from the perspective of the framework being integrated with a Russell-like language where the tasks of programming and proving are separated (see §3.3). However, the ideas we present will apply to other methods of program construction as well. As we explain in §5.4, the intended audience for this work is existing users of dependently typed programming languages.

The main features offered by our framework are as follows:

Error feedback: A testing tool is used to automatically identify errors that are indicated by unprovable proof obligations. When an unprovable proof obligation is identified, error feedback is provided to the user in the form of a counterexample description. This feedback is designed to give information that can be used to fix the error. We describe this error feedback and the design of the testing tool in Chapter 8.

Proof automation for user-defined properties: Generic heuristic-based proof automation is provided that is designed to be effective for discharging the proof obligations that arise from dependently typed programs. In particular, this automation makes use of the rippling technique [Bundy et al., 2005] and supports working with user-defined properties that involve inductively defined types and recursively defined functions. Moreover, we have focused on support for capturing program properties using subset types. We provide an in-depth description of this automation in chapters 6 and 7.

User hinting facilities: If a proof obligation cannot be discharged by the automation, the search tree of the failed proof attempt is shown to the user. The user can examine the search tree and attempt to help the prover by providing a hint. A hint takes the form of a conjecture that the proof automation will try to prove. If successful, the proof found is stored as a new lemma and the automation then tries to discharge the original proof obligation again with the help of this lemma.

In cases where the user gives a non-theorem as a hint, the testing tool is employed again to give counterexample-based error feedback. The above hinting mechanism is described in §7.13. This hinting mechanism gives an alternative to having to resort to a manual proof when the automation fails.

Lemma caching and lemma reuse: To make the proof automation more powerful and scalable, lemmas proven by the proof automation during proof searches are cached for reuse in future proofs. Several of our design choices center around the desire to cache lemmas that can be more easily reused. We give an overview of the lemma caching feature in §7.2.

Tactics for use in manual proofs: In cases where the proof automation fails and the user cannot help automate the proof by providing hints, the individual tactics that make up the proof automation can be usefully employed as part of manual proofs. These tactics are described in Chapter 7.

This framework is novel in that it presents a combination of integrated features that are not available in current dependently typed programming environments.

5.2 Components and Interactions

We now describe the components of the framework and explain how these are used to provide the features described above. The main components of the framework are as follows:

- The *testing tool* component is used to find counterexamples to proof goals where the counterexample descriptions are designed to be readable by the user. We employ a QuickCheck-like approach [Claessen and Hughes, 2000] for finding counterexamples.
- The *proof automation* component is composed of several tactics that are integrated to provide inductive proof automation. For example, we have designed tactics for simplifying goals, generalising goals [Aubin, 1976, Boyer and Moore, 1979, Aderhold, 2007] and performing rippling proofs [Bundy et al., 2005]. These tactics are structured using the Boyer-Moore theorem prover waterfall approach [Boyer and Moore, 1979].
- The *lemma database* component is used to store lemmas for use by the automation during proof attempts. Lemmas cached during proof search are stored here as well as the lemmas proven when the user supplies hints to the prover.

The following describes how the above components interact with each other when assisting the user in constructing a dependently typed program:

1. The user inputs a dependently typed function into the system in the style of a Russell program (see §3.3), where proof obligations are generated that must be discharged.
2. Generated proof obligations are sent to the proof automation component to be discharged.
3. The testing tool is employed by the proof automation to identify unprovable proof obligations as well as to detect overgeneralisations made during proof search. Moreover, the proof automation utilises the lemma database during proof search as a source of lemmas and as a place to cache lemmas.
4. There are then three possible forms of feedback that the framework can give to the user:

Success: If the proof automation can discharge all the generated proof obligations, the user is informed that their function has now been defined.

Error detected: If the testing tool generates a counterexample to any of the top-level proof obligations, the user is told an error has been found. A description of the counterexample found is displayed and the term in the body of the function that generated the unprovable proof obligation is identified to the user (see §3.3 for a description of how terms in the body of Russell functions can generate proof obligations). This information is intended to help the user identify and correct the error.

Proof automation failure: If the proof automation fails to discharge a proof obligation and the testing tool could not find a counterexample, the user will be shown a trace of the failed proof attempt. Assuming a proof is possible, the user can sometimes avoid having to resort to performing a manual proof by supplying a hint to the automation.

Figure 5.1 gives a high-level overview of how the framework components communicate and summarises how the user interacts with and gets feedback from the framework.

5.3 Usage Storyboards

To give a better understanding of the dialogue that is meant to take place between the user and the framework, we now present some typical usage scenarios. We describe how the user interacts with the system, how the system responds and how this feedback is used to construct a dependently typed program.

Correcting a Program Error

The following scenario involves the user correcting a program error:

1. The user enters a dependently typed function definition into the system.

2. When processing the function, the system generates several proof obligations. One of these proof obligations is unprovable because the function supplied contains an error.

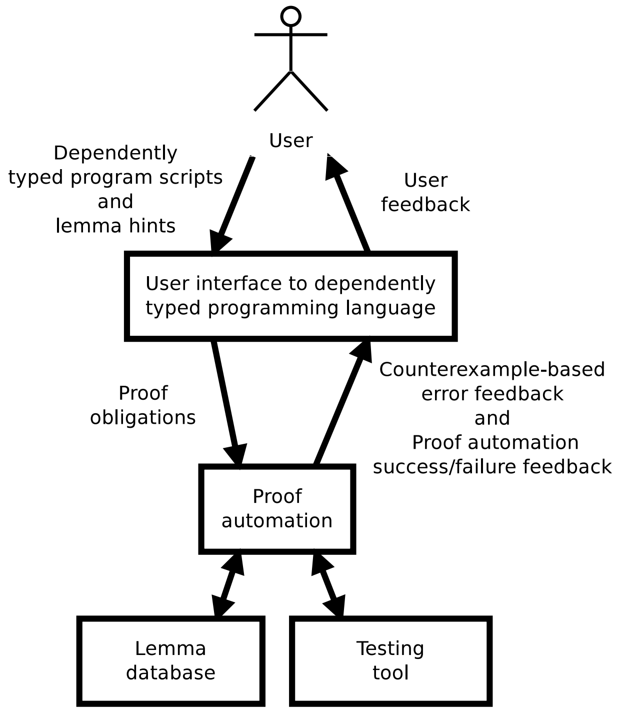

User
Dependently typed program scripts and lemma hints
User feedback
User interface to dependently typed programming language
Proof obligations
Counterexample-based error feedback and Proof automation success/failure feedback
Proof automation
Lemma database
Testing tool

Figure 5.1: High-level overview of the framework components and their interactions.

3. The testing tool identifies that one of the proof obligations is unprovable because a counterexample was found.
4. The user is presented with a description of the counterexample and is told which term and property in their program generated the unprovable proof obligation.
5. The user considers this feedback and uses the information given to identify and fix an error in the body of the function supplied previously.
6. Several proof obligations are again generated when the function is processed. This time, the proof automation is able to discharge all of these and the function definition is accepted.

Providing a Proof Hint

The following scenario involves the user providing a hint to help the proof automation solve a proof obligation:

1. The user enters a dependently typed function definition into the system.

2. When processing the function, the system generates several proof obligations.
3. The proof automation discharges all of the proof obligations except for one. The user is presented with a trace of the failed proof attempted.
4. The user identifies from the proof trace that a certain lemma could be useful in the proof attempt. The user inputs their proof hint by entering a conjecture that represents this lemma.
5. The automation proves the conjecture and adds the lemma proven to the lemma database.
6. The automation then reattempts the proof that failed previously. This time, with the help of the new lemma in the lemma database, the proof obligation is discharged and the function definition is accepted.

Notice here that, using proof planning terminology, the user is playing the role of a proof critic by suggesting how a failed proof can be patched (see §2.6).

5.4 Intended Audience

Our main audience is users of current dependently typed programming languages like Epigram, Agda, Coq and ATS. The features described should make program development easier and allow these users to be more proficient. Users of this audience who are familiar with formal proofs are likely to appreciate the lemma hinting mechanism and find the individual tactics useful for writing semi-automated proofs.

Ultimately, we would like to make programming with dependent types easy for users who have only had experiences with regular functional programming languages, such as Haskell, ML and OCaml. We believe these users, who are unlikely to have theorem proving experience, would benefit from the framework features we described, especially the improved proof automation. However, we note that members of this audience would need training in at least some aspects of formal proofs to make use of the lemma hinting feature.

We note that the features of the framework will also be useful to proof assistant users who wish to construct dependently typed definitions and, in particular, inductive proofs. Inductive theorem proving is a common tool in formal reasoning so improved automation here is likely to be appreciated. Likewise, testing tools are widely known to be useful for testing and refining conjectures during theory developments.

5.5 Prototype Implementation

We have built a proof-of-concept prototype to provide evidence that our framework can be used to make dependently typed programming more practical. This prototype is implemented within the Coq proof assistant, based around the Russell language. Specifically, the proof automation component of the framework acts upon the proof obligations generated from Russell programs.

Coq is a good choice as a foundation for demonstrating our ideas for the following reasons:

- The Russell language separates the tasks of programming and proving. This makes it easier to produce a prototype where the user interacts with the framework in the way that we have envisaged.
- Coq is a mature system with a large active user and development community. The practical benefit of this when developing a prototype is that there are many places to find help about Coq and there is lots of documentation available.
- Coq is packaged with many powerful tactics, such as decision procedures for linear arithmetic and propositional logic [Coq development team, 2006]. Moreover, Coq includes $\mathcal{L}_{\text{tac}}$, a domain specific language for writing new tactics that can make proof automation development easier [Delahaye, 2000].

In principle, systems such as Epigram, ATS and Agda could have been used to prototype our framework. However, these systems lack Coq's mature framework and level of built-in proof automation.

One possibility we considered was to develop our prototype in Isabelle so we could take advantage of Isabelle's existing rippling tactic [Dixon and Fleuriot, 2003]. The approach considered was to port Hurd's PVS-like predicate subtyping work from HOL [Hurd, 2001] for use in capturing program specifications. However, we felt that having to effectively design a new language in Isabelle would be more work and less straightforward than implementing rippling in Coq.

5.6 Conclusions

In this chapter, we introduced our framework for supporting dependently typed programming. The framework aims to make dependently typed programming more practical by providing assistance for coping with errors and constructing proofs. We have included features for identifying errors, giving feedback to errors, inductive proof automation for discharging proof obligations and a facility where the user can help the automation by providing high-level hints. In the next chapters, we describe the design of the framework components and their implementation details. We then present a series of case studies in Chapter 9 where we find that our prototype framework makes developing dependently typed functional programs significantly easier.

Chapter 6

Proof Patterns of Dependently Typed Programs

As described in the previous chapter, one important feature of our framework is to provide automation that is effective at discharging the proof obligations that arise from dependently typed programs. To implement suitable automation, we first need an understanding of the steps taken to discharge proof obligations manually. In this chapter, based on our own experiences of discharging proof obligations by hand, we describe these high-level steps in the form of *proof patterns*. The purpose of each proof pattern is to identify a pattern of proof that we need to provide automation for and to give an analysis of the situations where these patterns arise.

A proof pattern consists of the following: the features that a goal should have for a pattern to be applicable (the *pre-conditions*), the high-level proof steps that are carried out on the goal (the *description*) and a description of any notable features that the modified goal will have afterwards (the *post-conditions*). The pre- and post- conditions are intended to explain the rational behind how the proof patterns can be combined to describe the steps needed to discharge entire goals.

The proof pattern descriptions in this chapter were used as the foundation for the design of our proof automation. For each proof pattern in this chapter, we designed and implemented a tactic (described in the next chapter) that provides automation for that pattern of proof. For example, in this chapter, the purpose of the ripple proof pattern description is to identify the places where the rippling technique is applicable when discharging proof obligations (specifically, it is nonobvious that rippling applies in recursive call proof obligations). The ripple tactic in the next chapter gives a concrete implementation of a tactic that automates this proof pattern where, it should be noted, there are many design choices available in how a rippling tactic can be implemented.

6.1 The simplify Proof Pattern

A common proof step is to transform a goal into a simpler form before any complex reasoning techniques, like induction, are used. In particular, we find that top-level proof obligations can frequently be simplified for the following reasons:

- Goals that contain subset types can almost always be simplified by destructuring all subset type terms. Doing so gives access to the propositional parts of the subset type terms, which is usually needed to advance the proof.
- For each pattern matching equation x (see §3.3.1), x can almost always be used to rewrite the goal and then x can be discarded to make the goal simpler.
- Top-level proof obligations can usually be simplified by performing computations.

We now describe the simplify pattern:

Pre-conditions: None.

Description: The following describes the general steps used to simplify goals, where these steps are performed in a loop until no further progress can be made:

1. Simplify the goal using computation. For example, Coq's simpl tactic does this by applying appropriate reductions [Bertot and Castéran, 2004].
2. If the goal contains a subterm t with a type of the form $\{x \mid P\}$, destructure t into its computational part s and propositional part p. After doing this, p is accessible for use in the proof and terms that were of the form proj1_sig t can be reduced to s. These simplification steps can been seen in the example from §3.3.2.
3. For each assumption of the form H : x = t, where x is a variable and x is not a subterm of term t (i.e. a non-recursive equation), replace all occurrences of x by t and discard the assumption H. Each of these assumptions can be safely discarded after use because the variable replaced will have been eliminated from the goal (i.e. x has been substituted everywhere by

its definition). Using these assumptions in this manner can allow further simplification to take place. Pattern matching equations are generally non-recursive equations.

4. For each subterm in the goal of the form **match** x **with** ..., we can sometimes simplify the goal by destructuring x. This step performs case analysis on conditional statements, possibly producing subgoals. For example, x could be a boolean variable.
5. Rewrite the goal with equations that are known to be useful simplification rules. For example, it is common to rewrite occurrences of x ++ [] to x and occurrences of x $*$ 0 to 0.
6. Repeat the above steps until no further progress can be made.

Post-conditions: If any of the above steps applied, the resulting goal will generally be easier to prove than before.

6.2 The trivial Proof Pattern

Before attempting complex reasoning techniques like induction, it is usually sensible to first check if the goal is solvable by any standard automated tactics that are available. For example, base case proof obligations and the base cases of inductive proofs are sometimes solvable without performing induction. We now describe what we have named the trivial pattern:

Pre-conditions: None.

Description: The goal is proven using standard reasoning techniques such as propositional reasoning, proof by reflexivity or the application of a previously proven lemma.

Post-conditions: The goal is either discharged or unaltered.

6.3 The impossible_case Proof Pattern

The impossible_case pattern describes proofs where we must find a contradiction amongst the assumptions (in other words, reductio ad absurdum). Impossible case proof obligations (see §3.3.3) usually have this form. Moreover, the base cases of some inductive proofs have this form also. This proof pattern is described as follows:

Pre-conditions: There are no obvious ways to determine when this pattern applies. Although this pattern is always applicable when the conclusion is of the form False, this pattern can also apply when the conclusion does not have this form.

Description: The proof is completed by finding a contradiction amongst the assumptions. The method of doing this is influenced by representation choices. Some typical methods for showing contradictions are as follows:

- Propositional reasoning is used to show that the assumptions P and $\sim$P lead to a contradiction.
- We must sometimes prove the goal by reasoning that an assumption has a type that is uninhabited. For example, types like $0 = 1$, $h :: t = []$ and $0 \neq 0$ require such reasoning.

Post-conditions: The goal is either discharged or unaltered.

6.4 The induction Proof Pattern

Proof by induction is an essential tool for proving universally quantified statements about inductively defined data types and recursively defined functions. The proof obligations generated by the use of inductive families (see §3.3.1) and subset types (see §3.3.2) are always universally quantified statements and usually contain inductively defined data types and recursively defined functions. Thus, we find inductive reasoning common when discharging proof obligations. The induction pattern, which is used to begin an inductive proof, is as follows:

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

Description: Induction is performed using a suitable variable and induction principle. Inductive hypotheses can usually be made stronger by first making sure as many free variables as possible are universally quantified before performing induction (see §7.6).

Post-conditions: When induction is performed, base case and step case subgoals are produced. When standard induction principles are used, the inductive hypotheses in each step case are guaranteed to embed into the conclusion i.e. the rippling heuristic will be applicable to such subgoals (see §2.5).

6.5 The recursive_call Proof Pattern

The recursive_call proof pattern requires some analysis before it is presented. This pattern applies when a recursively defined function has a subset type as its output type. Assume that we are defining a dependently typed function that matches the following template, where function g has type T $\rightarrow$ T, P is a function that returns **Prop**, and $y_1 \ldots y_n$ are arbitrary terms:

```
Program Fixpoint f $x_1$ $x_2$ ... $x_n$ : {o:T | P o $x_1$ $x_2$ ... $x_n$} :=
  match ...
  | ...
  | ... ⇒ g (f $y_1$ $y_2$ ... $y_n$)
```

The term f y_1 y_2 $\ldots y_n$, which represents a recursive call to f, will generate a proof obligation because of the way subset types are used above (see §3.3.2). The first step of this proof pattern is to substitute with any pattern matching equations. The conclusion of the goal for this proof obligation will then have the following form:

```
P (g (proj1_sig (f $y_1$ $y_2$ ... $y_n$))) $x_1$ $x_2$ ... $x_n$
```

If the f y_1 y_2 $\ldots y_n$ term is destructured into its computational term f_s and propositional term f_p, and the proj1_sig is simplified away, the goal is transformed into the following form:

```
f_p : P f_s $y_1$ $y_2$ ... $y_n$
──────────────────────────
P (g f_s) $x_1$ $x_2$ ... $x_n$
```

Notice that the type of f_p and the conclusion term share syntactic similarities, where both contain the terms P and f_s. These similarities exists because the shape of both of these terms is determined by the output type of f.

6.5.1 Recursive Calls and Embeddings

In fact, it is common for an *embedding* (see §2.5) to exist between assumption f_p and the conclusion. The presence of an embedding is useful as rippling can then be used to guide the proof search.

An embedding will exist when, for all n, the n^{th} argument to P in f_p embeds into the n^{th} argument to P in the conclusion. The first argument will always embed in this scenario as f_s embeds into g f_s. The rest of the arguments will embed when, in reference to the program that produced the proof obligation, f was called recursively with argument y_n being a subterm of x_n, for each n. Many structurally recursive functional programs are defined using recursive calls that match this form. As such, we can expect embeddings to occur frequently in recursive call proof obligations and these embeddings can be used to guide proofs.

6.5.2 Example

We now reconsider the recursive call proof obligation generated from the function srev from §3.3.2. The shape of the step case of srev matches the description from the previous section and, as such, the recursive call proof obligation generated contains an embedding. The recursive call proof obligation can therefore be annotated as follows:

$$\frac{\mathsf{srev\_p} : \mathsf{length}\ \mathsf{srev\_s} = \mathsf{length}\ \mathsf{t}}{\mathsf{length}\ (\boxed{\boxed{\mathsf{srev\_s}}\ {+}{+}\ [\mathsf{h}]}) = \mathsf{length}\ (\boxed{\mathsf{h}\ ::\ \boxed{\mathsf{t}}})}$$

The rippling heuristic can then be used to determine what rules should be used to modify the conclusion such that srev_p can be used.

6.5.3 Multiple Recursive Calls

When a dependently typed function f contains multiple recursive calls in the step case, the recursive call proof obligation produced will contain multiple calls to t. Destructuring the result of each call to f produces a propositional term. By the same reasoning as before, it is possible for all of these propositional terms to embed into the conclusion. The proofs for such goals resemble the step case of inductive proofs that contain multiple inductive hypotheses.

6.5.4 Pattern Description

We now describe the recursive_call proof pattern:

Pre-conditions: The goal is a recursive call proof obligation (see §3.3.2) that was generated when defining some function f where f returns a subset type and the conclusion of the goal contains an occurrence of f.

Description: Firstly, pattern matching equations are substituted. Then the subset type result from each call to f is destructured into the propositional term p and computation term s, which then allows terms that have the form proj1_sig (f ...) to be simplified to s.

Post-conditions: The resulting goal is now likely to contain an assumption that embeds into the conclusion.

6.6 The ripple Proof Pattern

As we have seen, the recursive_call and the induction patterns can produce goals that contain embeddings. We can use rippling to guide the proof for such goals. The ripple proof pattern is described as follows:

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

Description: The rippling heuristic is used to apply proof steps that reduce differences between the embeddable assumptions and the conclusion. If all differences can be eliminated between the conclusion and the embeddable assumptions, the goal can be strongly fertilised. If only weak fertilisation is possible, the lemma calculation technique can be used to conjecture a missing lemma (see §2.6).

Post-conditions: Conjectures from lemma calculation are universally quantified statements about inductively defined types.

6.7 The cross_fertilise Proof Pattern

The cross_fertilise pattern describes a common pattern that arises when we write a program composed of several dependently typed functions that each have an output type of the form {x | P x = ... }, for some function P. This pattern involves the somewhat ad hoc usage of equations to forward the proof.

To describe this pattern by example, consider the following program, where srev is a weakly specified function that reverses a list and is defined in terms of sapp for appending lists:

```
Program Fixpoint sapp (A:Set) (a b:list A) :
  {r:list A | length r = length a + length b} :=
  (* ... *)
```

```
Program Fixpoint srev (A:Set) (a:list A) :
  {r:list A | length r = length a} :=
  match a with
  | []   ⇒ []
  | h::t ⇒ sapp (srev t) [h]
  end.
```

Notice that the return type of both sapp and srev has the following form:

```
{r:list A | length r = ... }
```

The recursive call proof obligation of the srev function has the following form after destructuring the recursive call and substituting the pattern matching equations:

```
srev_p : length srev_s = length t
________________________________________________
length ( proj1_sig (sapp srev_s [h] ) ) = length ( h :: t )
```

Following the ripple pattern, we can ripple out the RHS of the conclusion and weak fertilise with srev_p from right-to-left. If we then destructure the result from sapp, the goal has the following form:

```
sapp_p : length sapp_s = length srev_s + length [h]
________________________________________________________________
length sapp_s = S (length srev_s)
```

Notice that there are no embeddings to guide the use of sapp_p here. However, we are able to rewrite the conclusion using sapp_p from left-to-right to forward the proof. As the output type of srev and sapp share a common form, we can expect opportunities such as this when these functions appear in the same proof obligation.

As in the above, we generally find that making use of equations when there is some opportunity to do so is frequently useful. For example, in some situations, an available equational assumption could be used to rewrite another assumption instead of the conclusion. There will be situations when such an ad hoc approach is not productive but, when there are no embeddings to guide the proof, this seems a reasonable last resort. Simplifying goals by rewriting with available equational assumptions is a common strategy used in proof automation [Boyer and Moore, 1979, Kaufmann and Moore, 1997, Dixon, 2005].

The cross_fertilise pattern is described as follows:

Pre-conditions: The goal contains an assumption of the form H:s = t or H:t = s and the term s occurs elsewhere in the goal.

Description: Except in H, all occurrences of s are replaced with t. Assumption H is then discarded.

Post-conditions: H has been removed from the goal and all occurrences of s have been replaced with t.

6.8 The generalise Proof Pattern

We observe that there are frequently opportunities to generalise top-level proof obligations, such as by generalising common subterms, after simplification is performed. This pattern is seen in many places in our case studies (see Chapter 9). For example, we describe how common subterm generalisation is used on several occasions when discharging the top-level proof obligations generated when verifying a binary adder in §9.5.

As well as including common subterms, it is not uncommon for proof obligations to contain assumptions that are irrelevant to discharging the goal. For example, consider the following weakly specified program that inserts an item into a sorted list in sorted position where we use the function le_gt_dec : ∀ n m, {n ≤ m} + {n > m} to compare list items:

```
Program Fixpoint insert (x:nat) (a:list nat) :
  {r:list nat | length r = S (length a)} :=
  match a with
  | nil ⇒ [x]
  | h::t ⇒ if le_gt_dec x h then x::a else h::(insert x t)
  end.
```

The proof obligation generated by the term x::a in the step case has the following form:

```
Heq_a : h :: t = a
H : x ≤ h
______________________________________
length (x :: a) = S (length (h :: t))
```

Assumption H comes from performing pattern matching on the term le_gt_dec x h in the program. However, this assumption is not required for the proof as we are only interested in verifying the length of the list returned in this case. Likewise, when a function has a subset type term as an input parameter, each proof obligation will contain a corresponding assumption with that type. As with the above, this assumption may not always be needed to discharge each proof obligation.

When the proof of a goal is cached as a lemma, irrelevant assumptions can be problematic as these can make the lemma cached less general (see §7.2). Moreover, irrelevant assumptions can complicate inductive proofs (see §7.6).

6.8.1 Pattern Description

We now describe the generalise pattern:

Pre-conditions: Applies to any goal.

Description: The goal is generalised, such as by replacing common subterms by fresh universally quantified variables, generalising apart variables or eliminating irrelevant assumptions.

Post-conditions: The goal produced will be a more general version of the original. However, there is a danger of overgeneralising the goal.

6.9 Combining Proof Patterns

We find that the proofs required to discharge typical proof obligations can be described by a combination of the previously identified patterns. The following describes how these patterns can be composed to describe the general shape of the proofs that we want to automate:

- 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 ripple pattern is followed.

- Simplification followed by basic reasoning techniques will discharge some proof obligations. Proofs for base case proof obligations and base cases produced from performing induction typically have this shape. These proofs are described by the simplify pattern followed by either the trivial or impossible case pattern.

- For theorems that require induction to be performed, it is beneficial to have the current goal in its simplest and most general form first. This process can be described by following the simplify, generalise and then the induction pattern. Step cases of induction follow the ripple pattern.

In the next chapter, we present a concrete implementation of a tactic that automates proofs using the strategy described above. In Chapter 9, we evaluate this tactic against the proof obligations generated from a set of dependently typed programs, where we demonstrate that this tactic provides a high level of proof automation.

6.10 Conclusion

In this chapter, we have identified the patterns of proof that commonly emerge when discharging proof obligations that arise from dependently typed programs. We described how top-level proof obligations in particular benefit from being simplified and generalised before a proof attempt is made. For example, this is in contrast to the proofs automated by IsaPlanner, where top-level goals are not generalised before induction is performed [Dixon, 2005]. As IsaPlanner is typically used to prove theorem statements that are hand crafted by the user, the top-level goal is assumed to be in its most general form and therefore generalisation is not attempted.

We then remarked that inductive proofs are frequently required in practice, where the rippling technique can be used to guide the proof attempt for the step case [Bundy et al., 2005]. In recursive call proof obligations for programs that use subset types, we identified the non-obvious presence of embeddings in common situations. Rippling can thus be applied to guide these proofs also. In the next chapter, we describe tactics that are designed to automate the proof patterns described here so that practical support can be given for programming with dependent types.

Chapter 7

Automation of Proof Patterns

In the previous chapter, we described proof patterns that frequently occurred when discharging proof obligations generated from dependently typed programs. As part of our framework for supporting dependently typed programming, we now describe tactics designed to automate these patterns. These tactics have been implemented within the Coq proof assistant so, in Chapter 9, we can investigate the effectiveness of this automation using case studies. For each proof pattern in the previous chapter, we describe the design of a corresponding tactic with the same name in this chapter. For example, the ripple tactic provides automation for the ripple proof pattern.

The tactics presented in this chapter have been implemented in Coq using a combination of OCaml and Coq's tactic language $\mathcal{L}_{\text{tac}}$. We describe the high-level algorithms implemented for each tactic in this chapter as opposed to showing the actual code as the former is more concise and easier to understand for those unfamiliar with Coq tactic development.

With the exception of basic tactics common to most theorem provers, such as tactics for rewriting terms, generalising specified terms and performing structural induction, we name the important existing Coq tactics we have used to implement our tactics. For example, our simplify and trivial tactics provide their functionality by calling several nontrivial Coq tactics whereas our ripple, generalise and induction tactics are implemented using basic Coq tactics.

In addition to the design of these tactics, we also describe several extra features that have been added to make the proof automation a more practical tool. In §7.2, we describe our approach to caching lemmas found during proof search and ways that these lemmas can be reused by the tactics in future proof attempts. This is supplemented by a simple template-based technique for automatically conjecturing common forms of lemmas, such as commutativity, prior to proof attempts (see §9.7.3). Moreover, we employ heuristics for automatically identifying rules that can be used for simplifying goals (see §7.11). We then explain the feature of our automation that allows the user to provide hints to help the prover when proof search fails (see §7.13).

7.1 Top-Level Tactic Description

The top-level tactic that is invoked to automate proof obligations makes use of the Boyer-Moore theorem prover waterfall approach to structure calls to the tactics we have designed [Boyer and Moore, 1979]. In the waterfall approach, a fixed sequence of tactics is invoked on the current goal where the first tactic in the sequence is referred to as the top of the waterfall. When a tactic generates subgoals, each subgoal 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. Note that, as we describe later, the functionality of the cross_fertilise tactic has been merged into the simplify tactic and, likewise, the impossible_case tactic has been merged with the trivial tactic.

The top-level tactic thus performs the following steps for each goal:

1. The recursive_call tactic is invoked to destructure recursive calls. Recall that this can potentially produce assumptions that embed into the conclusion.
2. If an assumption embeds into the conclusion, the following steps are performed:
   (a) The simplify and trivial tactics are invoked in an attempt to discharge the goal trivially, where any changes made to the goal are undone on failure. When a proof is found here for a step case goal, this can indicate that induction was performed unnecessarily and that only case analysis was needed.
   (b) The ripple tactic is invoked, with backtracking occurring if ripple fails to fertilise the conclusion. Specifically, ripple must succeed for the next steps to be applied.
3. The simplify and trivial tactics are invoked.

4. The generalise tactic is invoked, with backtracking taking place if an overgeneralisation is detected. If the proof after this point fails, we allow backtracking to the point before generalise was invoked for cases where an overgeneralisation went undetected.

5. The induction tactic is invoked, with the top-level tactic being called on each subgoal generated. The intended behaviour here is that the ripple tactic, which is part of the top-level tactic, will exploit the presence of embeddings in step case goals. Subgoals generated here that contain embeddings are processed first because, as ripple must fertilise the goal before induction is performed again, we find this limits unproductive proof search.

Goals are processed in a depth-first search manner, with the top-level tactic taking a parameter that limits the number of times the induction tactic can be invoked on a sequence of subgoals to prevent looping (we use a default limit of 5). IsaPlanner's prover, which also makes use of rippling to automate inductive proofs, experiences similar looping behaviour [Dixon, 2005].

7.1.1 Relation to Proof Planning

Note that, although we make use of rippling and the lemma calculation technique as part of our proof automation, we do not use the proof planning approach (see §2.6) in this thesis. We would say that the proof planning approach was being used if each tactic was formalised as a method (i.e. with pre-conditions and post-conditions) and some reasoning was being performed by the machine to determine which combination of methods should be used to conduct the proof search. In our case, the same tactic is always used on every proof attempt.

7.2 Lemma Caching

Three databases are used to store cached lemmas, where the same lemma can be included in more than one database. Each database is intended to include lemmas that are suitable for use by certain tactics. For example, as we mention below, certain lemmas should not be utilised by the ripple tactic for efficiency and only lemmas that simplify the goal should be used by the simplify tactic. The databases used are as follows:

- The simplify lemma database contains directed equations for use by the simplify tactic. These lemmas are used when performing exhaustive rewriting to simplify the goal. In §7.11, we describe a heuristic that is useful for identifying obvious simplification rules for this purpose. For flexibility, we include commands for letting the user add rules to this database directly but the user is trusted to only add rules under which rewriting will terminate.

- The ripple lemma database contains directed equations that are used by the ripple tactic when performing rippling proof steps. 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 when meta-variables are absent [Bundy et al., 2005] we do not need to be concerned that some combination of cached lemmas might lead to non-terminating behaviour.

  However, for efficiency, an equation will not be added to the ripple lemma database for use 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 ∀ x, x = x + 0 and ∀ x, 0 = x ∗ 0 usually only serve to increase differences in rippling proofs when used from left to right.

- The trivial lemma database is used by the trivial tactic to automatically prove conjectures that are instances of goals that have been seen before. There are no restrictions on the contents of this database and all cached lemmas are added to this. To quickly determine if the current goal is an instance of a cached lemma, the standard technique of using discrimination trees (which are related to tries) is employed [Christian, 1993].

7.2.1 Irrelevant Assumptions and Caching Reusable Lemmas

In §6.8, we noted that top-level proof obligations can contain irrelevant assumptions. As proving goals with irrelevant assumptions can make the lemmas we cache less general, we need a strategy for handling such assumptions so that cached lemmas are useful in future proofs. For example, consider if we had to prove the following goal:

```
(x:nat) (y:nat) (z:nat) (H:y ≠ 0) ⊢ x + y = y + x
```

Assumption H and z are superfluous to proving this goal. If these assumptions are not discarded and we go on to prove the goal, the cached lemma will have the following form:

```
L : ∀ x y z, y ≠ 0 → x + y = y + x
```

As seen in the above, irrelevant subformulae can make a cached lemma less general as well as cumbersome to use. For example, the ripple tactic would only be able to use L as a rewrite rule when the y ≠ 0 side-condition was satisfied and some instantiation was given for z.

The problems irrelevant assumptions cause to lemma caching has also been identified in IsaPlanner but is not addressed there [Johansson, 2009, §5.6.1]. The following summarises the strategy that we use to eliminate certain irrelevant subformulae from cached lemmas:

- By examining the proof found for a goal, we identify which assumptions from the goal were not used in the proof so that we can then remove the corresponding irrelevant subformulae from the cached lemma. This technique, which we call *delayed generalisation*, is described in §7.10.
- Before performing induction, the induction tactic manipulates the goal so that potentially irrelevant assumptions do not form part of the inductive hypothesis when induction is performed (see §7.6). This avoids irrelevant assumptions being used unnecessarily in proofs, which can then allow delayed generalisation to eliminate such assumptions when the proof is finished.

We also investigate heuristics for removing irrelevant assumptions during the proof attempt as part of generalisation (see §7.5.5), which is a step commonly seen in other generalisation algorithms [Aderhold, 2007, Boyer and Moore, 1979].

7.3 The simplify Tactic

We now begin our discussion of the tactics we have designed for automating proof patterns, starting with the simplify tactic (i.e. which automates the simplify proof pattern from §6.1). The simplify tactic applies the following steps in sequence and repeats until no progress is made:

Subset types: All subset type terms in the goal are destructured.

Reductions: The goal is simplified using Coq's simpl tactic, which simplifies the goal by performing computations [Bertot and Castéran, 2004].

Conditional statements: 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 H : x = t, where x is a variable and x is not a subterm of term t (i.e. a non-recursive equation), we replace all occurrences of x by t and discard H. This is implemented with Coq's subst tactic [Bertot and Castéran, 2004].

Injectivity: Equational assumptions are simplified using the knowledge that constructors are injective functions. For example, given H : cons h t = cons 0 nil, we can generate the assumptions h = 0 and t = nil and discard H. We use Coq's injection tactic for this [Bertot and Castéran, 2004].

Rewriting: The goal is exhaustively rewritten with cached lemmas that have been selected for use as simplification rules (see §7.2). A conditional rewrite rule can only be used when the subgoals it generates are discharged by the trivial tactic.

Use equational assumptions: The cross_fertilise tactic is called to rewrite with any equational assumptions available.

Removal of Non-informative Equations: We automatically discard assumptions of the form x = x from the goal. It is not uncommon for such equations to be introduced when performing case splits and these equations only serve to clutter goals. Of course, non-informative assumptions can have other forms but we have only considered those of the form x = x so far in this work.

7.4 The trivial Tactic

The trivial tactic is intended to automate the trivial proof pattern as well as the impossible_case pattern. We decided against implementing a separate tactic for each of these patterns for efficiency reasons as a propositional logic decision procedure is naturally required to automate both patterns.

The trivial tactic attempts the following procedures in sequence:

Lemma cache: If the goal matches a lemma from the lemma database used by this tactic, the lemma is used to prove the goal. The use of the symmetry property of equality is used so that, for example, a lemma of the form s = t will match a goal of the form t = s as well as s = t.

Decision procedures: We use Coq's intuitionistic propositional logic decision procedure (tauto) to attempt to prove the goal [Bertot and Castéran, 2004].

Impossible cases: When the goal contains an assumption that has an uninhabited type, such as the type h :: 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 [Bertot and Castéran, 2004].

7.5 The generalise Tactic

In this section, we describe the design of the generalise tactic. This tactic makes use of several heuristics to generalise goals automatically.

7.5.1 Overview: An Aggressive Generalisation Algorithm

In contrast to more cautious approaches where generalisation is only used as much as is needed to allow the proof to succeed, our tactic generalises more aggressively. For example, the generalisation heuristics in Verifun will only generalise common subterms that occur in recursive positions [Aderhold, 2007] whereas we always attempt to generalise all common subterms. Generalising more aggressively has the following benefits:

Reusable lemmas: The primary reason for generalising more aggressively is that this results in more general lemmas being cached, where such lemmas are then more reusable in future proofs.

Efficiency: The proof search space for generalised lemmas tends to be smaller as generalising will often make the goal simpler and, for example, reduce the number of variables that are available for induction to be performed on.

Conciseness: The proofs of more general lemmas tend to be more concise, less cluttered and easier to read. This is particularly important when we want the user to examine failed proof traces when providing hints to the prover.

Our generalisation algorithm is based on the common subterm generalisation algorithm used in IsaPlanner [Dixon and Fleuriot, 2004]. During development, we added additional generalisation stages to the algorithm. IsaPlanner's generalisation algorithm only generalises common subterms whereas our algorithm also generalises by inverse functionality, generalises apart and attempts to eliminate irrelevant assumptions.

We now give an overview of our generalisation algorithm, where we make use of the terminology introduced in §2.8. The algorithm generalises the current goal by performing the following step in sequence:

1. Generalise by inverse functionality (see §7.5.2).
2. Generalise common subterms (see §7.5.3).
3. Generalise apart (see §7.5.4).
4. Eliminate irrelevant assumptions (see §7.5.5).
5. Check for overgeneralisations (see §7.5.6).

We explain the details of these stages in the following sections.

7.5.2 Step 1: Inverse Functionality

In its most general form, inverse functionality can be used to generalise statements of the form $\mathsf{f}\ \mathsf{x}_1 \ldots\ \mathsf{x}_n = \mathsf{f}\ \mathsf{y}_1 \ldots\ \mathsf{y}_n$ by removing the application of f to produce the statement $(\mathsf{x}_1 = \mathsf{y}_1) \wedge \ldots \wedge (\mathsf{x}_n = \mathsf{y}_n)$. We find that naive use of inverse functionality frequently leads to overgeneralisations so we are more cautious in this stage than others. For this reason, we restrict the use of inversion functionality to cases where $\mathsf{n} = 1$. For example, this strategy will successfully generalise in the following cases:

rev (x ++ y) = rev (rev ((rev y) ++ (rev x)))	generalises to	x ++ y = (rev ((rev y) ++ (rev x)))
length (x ++ (y ++ z)) = length ((x ++ y) ++ z)	generalises to	x ++ (y ++ z) = (x ++ y) ++ z
S (x + 1) = S (1 + x)	generalises to	x + 1 = 1 + x
x ++ (x ++ y ++ z) = x ++ ((x ++ y) ++ z)	generalises to	x ++ y ++ z = (x ++ y) ++ z
x ∗ (y + z) = x ∗ (z + y)	generalises to	y + z = z + y

Note that in, for instance, the last example, generalisation by inverse functionality on $\mathsf{x} * (\mathsf{y} + \mathsf{z}) = \mathsf{x} * (\mathsf{z} + \mathsf{y})$ is allowed as f is taken as the curried function mult x (i.e. which only has one parameter).

As with all generalisation heuristics, there are cases where the generalisations made will be productive (as shown above) and in other cases overgeneralisations can occur. For example, the goal length x = length (rev x) would be overgeneralised to x = rev x by inverse functionality. Such overgeneralisations would be identified by the counterexample finder.

7.5.3 Step 2: Common Subterm Generalisation

In IsaPlanner, it was found that generalising all maximal common subterms in a goal, where terms of higher-order type are not treated as generalisation candidates, was a successful strategy to use when performing lemma calculation [Dixon, 2005]. We use the same strategy when generalising Coq terms. Our algorithm performs as follows:

1. The set of all subterms s that occur more than once in the conclusion is generated.
2. A subterm t from s is generalised if the following criteria is satisfied:
   (a) The term t is not a subterm of any of the other terms from s.
   (b) The type of t is not **Prop** (e.g. int $\rightarrow$ int and x + 0 = x have this type) or **Set** (e.g. nat has this type). This criterion prevents generalising terms of the form fun x $\Rightarrow$ ... and type variables.

Notice that we allow constants to be generalised. Such generalisations can sometimes be important in rippling proofs [Ireland, 1995].

7.5.4 Step 3: Generalising Apart

When the conclusion is an equation, a variable x is generalised apart if it occurs at least twice on each side of the equation and occurs the same number of times on both sides of the equation. We require the latter condition as, when generalising apart, we always simultaneously generalise one occurrence of x from the LHS along with one occurrence of x from the RHS as opposed to generalising two occurrences of x on just one side of the equation.

Generalising apart x occurs 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 strategy will successfully generalise apart the following equations:

x + (x + x) = (x + x) + x	generalises to	a + (b + c) = (a + b) + c
length (x ++ x) = length x + length x	generalises to	length (a ++ b) = length a + length b
x ∗ (y + y) = x ∗ y + x ∗ y	generalises to	x ∗ (a + b) = x ∗ a + x ∗ b
max x (max x y) = max y (max x x)	generalises to	max a (max b y)= max y (max a b)

7.5.5 Step 4: Eliminating Irrelevant Assumptions (the irrelevance Tactic)

We now describe an algorithm for eliminating assumptions that are likely to be irrelevant to proving the current goal. This is implemented as a tactic named the irrelevance tactic. This tactic is also useful in manual proofs as it can help make goals more readable by discarding assumptions that only serve to obfuscate the goal. The algorithm used has similarities to the irrelevance heuristic from the Boyer Moore theorem prover, where variable sharing between terms is considered to determine relevance [Boyer and Moore, 1979].

The irrelevance tactic works by recursively marking assumptions that it guesses are "probably relevant" to proving the goal. When finished, all the assumptions that have not been marked are discarded, where the tactic is guessing that the discarded assumptions are irrelevant. The irrelevance tactic operates as follows, where the special treatment of assumptions of type **Set**, **Prop** and **Type** is explained after:

1. Initially, no assumptions are marked as probably relevant.
2. With the exception of assumptions that have type **Set**, **Prop** and **Type**, we recursively reclassify assumptions according to the following criteria until no further reclassifications occur:

   R1: All assumptions that occur in the conclusion are probably relevant.

   R2: Assumption x is probably relevant if y : t is a probably relevant assumption and x occurs in t.

   R3: Assumption x : t is probably relevant if a probably relevant assumption occurs in t.

3. Assumptions that have type **Set**, **Prop** and **Type** which occur in the type of a probably relevant assumption are marked as probably relevant.
4. All assumptions that are not marked as probably relevant are discarded

We now give an example that has been constructed to show how assumptions are incrementally classified by the rules above. Consider the following goal which contains several irrelevant assumptions:

```
(w x y z:nat) (H1:w = y) (H2:y = 1) (H3:z = 0) ⊢ S x = x + w
```

The algorithm above correctly identifies that only the assumptions z and H3 are irrelevant with the following reasoning:

1. x and w are probably relevant because they occur in the conclusion (R1).
2. H1 is probably relevant because w is probably relevant and w occurs in the type of H1 (R3).
3. y is probably relevant because y occurs in the type of the probably relevant assumption H1 (R2).
4. H2 is probably relevant because y is probably relevant and y occurs in the type of H2 (R3).

To explain the special treatment of assumptions of type **Set**, **Prop** and **Type**, consider the following goal where y is an irrelevant assumption and the, usually implicit, type parameter for length is shown:

```
(A:Set) (x y:list A) ⊢ length A x = length A x + 0
```

If assumptions of type **Set** were considered by rules R1, R2 and R3, y would be incorrectly marked as probably relevant by rule R3 because A is used in the conclusion and A occurs in the type of y.

7.5.5.1 Limitations

We note here some cases that demonstrate the limitations of the irrelevance tactic:

Overclassifications: Relevent assumptions that do not share variables with the conclusion are never marked as probably relevant. For example this would happen in the following goals, where H is relevant to proving the goal in each case by showing a contradiction exists:

(x:nat) (H:0 $\neq$ 0) $\vdash$ x + 1 = x

(x y:nat) (H:y $\neq$ y) $\vdash$ x + 1 = x

However, the examples above are not an issue in practice as we would usually expect the impossible_case tactic to discharge such goals before generalisation is attempted.

Underclassifications: Irrelevant assumptions that share variables with the conclusion will always be marked as probably relevant. For example, this will happen in the following goal, where H is irrelevant but is incorrectly marked as probably relevant:

```
(x:nat) (H:x ≠ 0) ⊢ x + 0 = x
```

It is unclear how the irrelevance tactic could be extended to generalise correctly in the above case and not risk overgeneralising in others. For instance, H is correctly identified as relevant by our tactic in the following similar looking goal:

```
(x:nat) (H:x ≠ 0) ⊢ (x − 1) + 1 = x
```

Note that the minus operator for natural numbers is defined in Coq in such a way that 0 − 1 = 0. Assumption H is therefore relevant here as discarding H would make the goal unprovable.

To generalise correctly in both of these examples, some domain specific knowledge or analysis of how the functions being used are defined would be needed. One approach to identifying irrelevant assumptions here would be to only allow an assumption to be discarded as irrelevant if a counterexample check determined that doing so would not transform the goal into a nontheorem. This approach has been used in Verifun [Aderhold, 2007].

7.5.6 Step 5: Checking for overgeneralisations

We use the typical approach of detecting overgeneralisations by using a testing tool. For simplicity, we only test for overgeneralisations after all generalisation steps are made as opposed to testing after each generalisation step. For this, we use the QuickCheck-like testing tool we have developed for Coq (see Chapter 8). With this tool, we find that, as long as a goal does not contain any difficult to satisfy propositional assumptions, a small number of tests (e.g. 10) are sufficient to detect overgeneralisations in most cases.

7.5.7 Unblocking Rippling by Generalising Apart

We incorporated the heuristic for generalising apart into the generalise tactic when we observed that rippling proofs tend to fail when induction is performed on a goal where generalisation apart is applicable. For example, when only using basic definitions, the rippling proof for the theorem x + S y = S (x + y) is trivial but the proof of x + S x = S (x + x) is problematic. After performing induction on x on the latter statement, rippling becomes blocked in the step case and lemma calculation does not apply. This problematic theorem is given as an example of a rippling proof that can be unblocked by discovering the wave rule ∀ x y, x + S y = S (x + y) using a lemma speculation critic [Ireland and Bundy, 1996, theorem T15]. Johansson shows in IsaPlanner that rippling can also succeed here without lemma speculation if lemma synthesis is first used to find the same lemma prior to the proof attempt [Johansson, 2009, p114].

We advocate that a conceptually simpler and more natural technique is to generalise apart where possible before performing induction. In this example, the occurrences of x in the top-level goal x + S x = S (x + x) are generalised apart by the generalise tactic to give x + S y = S (x + y), which is then trivial to prove by induction.

7.6 The induction Tactic

We now discuss the induction tactic, which initiates an inductive proof.

7.6.1 Inductive Variable and Induction Principle Choice

To start an inductive proof, we must pick a variable on which to perform induction 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. When the induction tactic is invoked in the top-level tactic (see §7.1), the subgoals produced are processed by another call to the top-level tactic. If either of these subgoals cannot be discharged, backtracking occurs to the point where induction was performed. The induction tactic then performs induction on the next variable in the variable list and the top-level tactic is invoked on these subgoals. This continues until the contents of the variable list is exhausted.

We find this naive approach to selecting the induction variable and induction principle performs well enough in practice. If an unproductive induction variable is chosen, backtracking tends to occur quickly as the `ripple` tactic is unable to make any progress. A similar approach is used in IsaPlanner when performing induction and is found to perform well in practice as well [Dixon, 2005].

7.6.2 Modifying the Conclusion Before Performing Induction

Before performing induction, it is usually advantageous to modify the conclusion so that certain variables are universally quantified. Quantifying variables can strengthen inductive hypotheses and gives more opportunities for rippling to fertilise in step cases. For example, consider the following two goals, where each goal can be transformed into the other with appropriate universal introduction and reintroduction:

1. (x:nat), (y:nat) ⊢ x + y = y + x
2. (x:nat) ⊢ ∀ (y:nat), x + y = y + x

Performing induction on x in the first goal would result in a weaker induction hypothesis than performing induction on x in the second goal as y would not be universally quantified in the former. We use the following procedure before performing induction on a variable x, where the exceptions to what assumptions we reintroduce are explained in the following sections:

1. Exhaustively perform universal introduction.
2. Reintroduce each assumption into the conclusion that matches the following criteria: the assumption is not the induction variable x, the assumption type is anything except for **Prop** and the assumption does not occur in the conclusion.

   Note that, given some assumption P is defined in terms of another assumption Q, P has to be reintroduced before it is valid to reintroduce Q. Apart from cases such as this, the order of reintroduction is unimportant. For example, ∀ x y, R x y is equivalent to ∀ y x, R x y and there is no reason that the tactics we use should prefer one of these forms over the other.

7.6.2.1 Treatment of Irrelevant Assumptions

We prevent the reintroduction of variables that do not occur in the conclusion to avoid complicating inductive hypotheses when irrelevant assumptions are present. For example, if induction is performed on x in the goal (x:nat) $\vdash$ $\forall$ (y:nat), x + 0 = x, where y is irrelevant, the inductive hypothesis is needlessly complex and requires we instantiate y during fertilisation.

Likewise, reintroducing propositional assumptions (i.e. type **Prop**) can make an inductive proof more complex than necessary. Consider the following variants of a goal where y $\neq$ 0 is irrelevant in both:

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

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 complex to discharge. When induction is performed on the first goal, the step case inductive hypothesis will contain an implication. In general, step cases of this form require piecewise fertilisation [Armando et al., 1999] to prove which means that the rippling tactic used must be more sophisticated.

By not reintroducing propositional assumptions before performing induction, we thus avoid complicating some proofs when goals contain irrelevant assumptions. The above strategy avoids making use of certain irrelevant assumptions in a proof, which then allows our delayed generalisation algorithm to identify and eliminate these assumptions later when caching lemmas (see §7.10).

As we note in §10.4.6, we currently do not support piecewise fertilisation [Armando et al., 1999], which will be needed to solve inductive proofs where an implication appears in the inductive hypothesis. As such, when relevant or irrelevant assumptions appear as implications in the inductive hypothesis, the ripple tactic will fail.

7.7 The recursive_call Tactic

This recursive_call tactic is straightforward to implement and follows the steps described in §6.5.

7.8 The ripple Tactic

We now discuss the ripple tactic for automating proofs with rippling. This is largely based on the implementation of dynamic rippling in IsaPlanner [Dixon, 2005, Johansson, 2009]. We first give a high-level overview of the ripple tactic, where each stage described below is elaborated on over the next few sections. The ripple tactic works as 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. This is intended to automatically identify inductive hypotheses and other assumptions that embed. The restriction on the type of the assumption is used to prevent, for example, the assumption H : list nat from being considered as a given. Treating such assumptions as givens is rarely useful and needs to be avoided as these forms of assumption occur frequently in the conclusion as type variables.

2. The tactic generates all the ways that the current goal conclusion can be modified using available equational lemmas (see §7.8.4). The list of equational lemmas used is initially populated with equations generated from function definitions (see §7.8.1). Only modifications that reduce differences in the conclusion with respect to the list of givens are allowed.

   Depth-first-search is then used to explore the search space. We limit the depth of the search space to 10 so that, in cases where it will not be possible to fertilise the goal, the proof will fail faster. This limit seemed reasonable as the longest rippling proof found with our prover when ran against a theorem corpus from IsaPlanner (see §7.14) involved 4 rippling proof steps.

3. Fertilisation is attempted when no further difference reducing transformations can be found (see §7.8.1). We do not allow backtracking on the way weak fertilisation is performed as we find the choice is typically unimportant to a proof. Likewise, givens are rarely useful after weak fertilisation and so are discarded afterwards.

7.8.1 Generating Equations from Function Definitions

When performing rippling proof steps, we make use of equations that are generated from function definitions. For example, the standard functions plus and max (see §A.1) can be represented with the following two equations, where each equation targets one pattern matching clause from the function definition:

```
plus_base  : ∀ m,         plus 0 m = m.
plus_step  : ∀ p m, plus (S p) m = S (plus p m).

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. The form of these equations is similar to the form seen in other presentations of rippling [Dixon, 2005]. 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 implemented an algorithm in Coq that will automatically generate these forms of equations from structurally recursive functions, where this algorithm is sufficient for all the examples considered in this thesis. This generation process is best explained by example. Consider the following definition of plus:

```
Fixpoint plus (n m:nat) : nat :=
  match n with 0 ⇒ m | S p ⇒ S (plus p m) end.
```

We first transform the function definition into an equational goal. For plus, we generate the following goal:

```
∀ n m, plus n m =
  match n with 0 ⇒ m | S p ⇒ S (plus p m) end.
```

The goal, which is provable by reflexivity, is derived from a simple syntactic transformation of the original function definition: the LHS consists of the function name and parameters, the RHS consists of the body of the function and the variable names for the input parameters of the function are universally quantified.

To generate the defining equations, exhaustive universal introduction is performed and case splitting occurs on the recursion variable. This produces subgoals, where each corresponds to an equation we want to generate. Each subgoal is proven by reflexivity, where each goal and its proof are cached as a lemma. Delayed generalisation is used to remove any unnecessary variables from these cached lemmas (see §7.10).

The reason the equation generation process is implemented using tactics is because this was deemed the simplest implementation approach. Specifically, each stage in the generation process is trivial to implement with standard tactics.

7.8.2 Rippling Annotations

After transforming the conclusion, we generate all first-order rippling annotations with respect to the list of givens. A rippling annotation is represented as a regular Coq term, where identity functions are used to decorate terms in the traditional way to represent annotation features such as wave fronts and holes [Basin and Walsh, 1996].

For example, to represent a wave hole we introduce the function wave_hole which is defined as fun (A:**Type**)(x:A) ⇒ x. We can then represent that some subterm t in a term is a wave hole by replacing t with wave_hole t. We can introduce functions to represent inward wave fronts, outward wave fronts and sinks in the same fashion, where these can then be used to annotate Coq terms with rippling annotations.

Due to time constraints, our function for calculating annotations does not look inside the following term constructs when searching for differences between terms: **let** x := ... **in** ..., fun x ⇒ ... or **match** x **with** .... For example, when calculating embeddings, the term t will embed into the term fun x ⇒ s only when both of these terms are syntactically the same, given any term for s.

As seen in previous work, special treatment is needed to annotate goals that contain higher-order features such as λ expressions [Smaill and Green, 1996, Dixon, 2005]. However, we can still support rippling proofs that include λ expressions as long as the differences in the conclusion do not appear inside these.

7.8.3 Ripple Measures

To check if a modification to the conclusion is difference reducing, we use the sum of distances ripple measure [Dixon, 2005]. 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.

When the conclusion can be annotated in multiple ways, it can have multiple measures. In such cases, we use Dixon's notion of a threshold measure to efficiently decide when conclusion transformations should be allowed [Dixon and Fleuriot, 2004, §7.10].

7.8.4 Conclusion Transformations

When searching for ways to transform the conclusion, we consider every way the conclusion can be modified by only rewriting one subterm. For example, given commutativity of + and the conclusion is (a + b) + c = a + b, we would want to generate the transformations (b + a) + c = a + b, c + (a + b) = a + b and (a + b) + c = b + a. We only allow conditional rewrite rules to be applied when the side-conditions can be discharged by calling simplify and trivial in sequence. As with IsaPlanner, the state of the conclusion is stored each time a transformation is made and we backtrack if the same state is seen again during search.

7.8.4.1 Controlling Case Splits with Rippling

We make use of a similar technique to IsaPlanner to control case splitting during rippling proofs [Johansson, 2009]. Briefly, before checking if a conclusion transformation is measure reducing, when the conclusion contains a subterm of the form **match** x ..., a case split is automatically performed on x. In each subgoal produced, for any new assumption that are introduced of the form H : x = t, where x is a variable and x is not a subterm of term t, substitution of x is performed using H and then H is discarded. 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 the remaining subgoals.

7.8.5 Weak Fertilisation

There are usually choices in the way a conclusion can be weak fertilised. The general heuristic for weak fertilisation, that we use, is that the LHS of the conclusion should only be rewritten by using a given from left-to-right and the RHS of the conclusion should only be rewritten by using a given from right-to-left [Boyer and Moore, 1979, Dixon, 2005].

To weak fertilise with a given, we first attempt to rewrite the LHS of the conclusion and, on failure, attempt to rewrite the RHS of the conclusion. For multiple givens, we repeat this procedure for each given individually, where weak fertilisation only succeeds when all givens can be used.

7.9 The cross_fertilise Tactic

We now explain the implementation of the cross_fertilise tactic. For each assumption H in the goal with the form $\forall$ x_1 … x_n, s = t, we perform the following procedure:

1. For each assumption P, we attempt to rewrite P with H used as a left to right rewrite rule. This is repeated for the conclusion.

2. If no terms were rewritten in the previous step, attempt the rewriting operation again using H as a right to left rewrite rule.

3. If any terms were rewritten in the previous two steps, discard H.

This tactic will produce subgoals when H is a conditional rewrite rule. Note that the tactic always terminates because, whenever terms are rewritten, the number of assumptions in the goal decreases and this can only happen a finite number of times.

Recall that the ripple tactic discards inductive hypotheses after fertilisation is performed (see §7.8.5). This step is important when the cross_fertilise tactic could be called as calls to this can undo the progress made by the weak fertilisation step of the ripple tactic.

7.10 Delayed Generalisation

Now that we have described all of the tactics that make up our automation, we move on to describing additional features that concern lemma caching and the ability to give hints. In this section, we describe an algorithm that we have named *delayed generalisation* that is used when caching lemmas. Given a lemma statement P and its proof t, the purpose of this algorithm is to produce a more general lemma by inspecting both P and t to identify irrelevant subformulae that can be safely removed.

As an example, consider a lemma of the form $\forall$ x y, y $\neq$ 0 $\rightarrow$ x + y = y + x, where the proof of this lemma did not make use of the witness for y $\neq$ 0. By inspecting the lemma statement and the proof, delayed generalisation can produce a more general lemma of the form $\forall$ x y, x + y = y + x. This offers an alternative to eagerly guessing which assumptions are irrelevant and removing them in the middle of a proof [Aderhold, 2007, Boyer and Moore, 1979], which can cause overgeneralisations to occur.

However, note that delayed generalisation would not, for example, be able to remove the y $\neq$ 0 assumption from the lemma above if the assumption had been needlessly used in some way in the proof. For future work, it would be useful to explore how proofs could be simplified to reduce the unnecessary use of assumptions before delayed generalisation is applied.

7.10.1 Illustrative Examples

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

```
∀ (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 y ≠ 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. When one of the λ terms at the start of t introduces a variable that is not used in the body of t, this represents an assumption that was not required to construct the proof. In this case, variables z and H are not used in the proof and are thus superfluous to the lemma statement.

A special case to be aware of is that universally quantified variables that occur in the conclusion of P should always be retained when generating t' : P'. For example, consider the case where P is ∀ n, 0 ∗ n = 0. A standard proof t for this lemma in Coq is fun n ⇒ refl_equal 0. As the variable n is not used in t, it is nonsensical to eliminate the corresponding variable n from P. For this reason, the delayed generalisation algorithm never eliminates universally quantified variables that occur in the conclusion of P when generating t' : P'.

7.10.2 Algorithm Description

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 its proof t was constructed by first performing exhaustive universal introduction. Under these assumptions, term t will have the 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.

The above is implemented in Coq as a command that, when supplied with a theorem, will attempt to derive and store a more general version of that theorem using delayed generalisation.

When our automation finishes constructing a proof t for a goal g and t and g are cached as the lemma P, delayed generalisation is used on P to produce P'. P' is then used to prove g. This step is important because if P is used to prove g, P will be instantiated with, 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 further 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.

7.11 Automatic Identification of Simplification Rules

To provide better support for working with definitions introduced by the user, we describe two simple heuristics which we have found to be successful for automatically identifying equational lemmas that can be productively used by the simplify tactic. For example, rewriting from left to right with the rules ∀ a, a ++ [] = a and ∀ a, rev (rev a) = a can be useful for simplifying goals. Simplification rule sets are normally hand chosen based on experience and users are trusted to choose rules that do not cause simplification tactics to loop. As we discuss in the next section, the power of our proof automation can be increased by supplying the simplify tactic with appropriate simplification rules. Similarly, we note that the simplification tactic in ACL2 [Kaufmann and Moore, 1997] contributes significantly to the power of the prover.

7.11.1 Motivation

In the following, we describe several reasons why we found simplification rule detection an important feature to add to our framework:

Increasing proof coverage: Appropriate use of simplification can allow a theorem to be solved trivially without induction and rippling. For example, consider if we were required to prove ∀ a, length (rev (rev a)) = length a and had access to the lemma L : ∀ a, rev (rev a) = a. The goal can be solved easily by rewriting the goal using L from left to right and then finishing with a proof by reflexivity. This avoids the more complex approach of performing a proof by induction. Moreover, in cases where the inductive proof is difficult and rippling can become blocked, simplifying the goal first can make the inductive proof more likely to succeed.

Producing Reusable Lemmas: Neglecting to apply obvious simplification rules before performing induction can result in less reusable lemmas from being cached. For example, consider the following goal:

```
∀ a b c, (a ++ b ++ []) ++ c = a ++ (b ++ c)
```

Given that the prover knows the lemma L : ∀ a, a ++ [] = a, performing simplification with L before proving the goal by induction results in the general lemma ∀ a b c, (a ++ b) ++ c = a ++ (b ++ c) being cached. Alternatively, proving the top-level goal directly by induction leads to a less general lemma being cached. Additionally, as discharging a single goal can involve several lemmas being cached in the process, simplification of the top-level goal can also make these additional lemmas more general also.

Efficiency: Simplification can make proof search more efficient in that costly inductive proof attempts can be avoided entirely and, when induction is needed, simplification can reduce the proof search space by removing variables and terms from the goal.

7.11.2 Heuristics for Identifying Simplification Rules

We now describe heuristics for identifying simplification rules. Whenever a lemma is cached, these heuristics are used to automatically identify lemmas that are appropriate for simplifying goals. Suitable lemmas are added to the lemma database used by the simplify tactic. Recall that the simplify tactic will exhaustively rewrite the goal with all equational lemmas in its lemma database, with the restriction that any side-conditions produced during rewriting must be solved with a call to the trivial tactic.

7.11.2.1 Basic Terms

We introduce the notion of a *basic term* to describe terms that intuitively cannot be simplified any further. A basic term is defined recursively as a term of the form f $x_1 \ldots x_n$ where the following is true:

1. The term f is either a constructor (e.g. S or cons), a type constructor (e.g. list) or an inductive data type (e.g. nat).
2. Individually, $x_1 \ldots x_n$ are either basic terms or have the type **Type** or **Set**.

In other words, a basic term can only include variables that act as type variables and the only function symbols that are allowed are constructors. For example:

- The terms O, (S O), nil nat and cons nat O (nil nat) are basic terms.
- Given A:**Type**, the term nil A is a basic term.
- The term 0 + 0 is not a basic term because + is not a basic term.
- Given x:nat, the terms S x and cons nat x (nil nat) are not basic terms because x does not have the type **Type** or **Set**.

7.11.2.2 Basic Terms Heuristic (BH)

The first heuristic we use for identifying simplification rules is as follows:

> **BH:** The lemma ∀ x_1 ... x_n, s = t should be used as a left to right rewrite rule for simplification when the term s in the lemma statement is not a basic term and the term t in the lemma statement is a basic term.

Intuitively, this rule says that we can simplify a goal by using any lemma that can be used to replace non-basic terms with basic terms. The following are some example lemmas that are identified as left to right rewrite rules for simplification by BH:

```
∀ x, x − x = 0
∀ x, x ∗ 0 = 0
∀ x, 1 ^ x = 1
∀ x, min x 0 = 0
∀ x, length x = 0 → x = []
```

We find this heuristic selects obvious simplification rules as it is almost always beneficial to eliminate variables in this manner from a goal.

7.11.2.3 Embeddings Heuristic (EH)

The second heuristic we use for identifying simplification rules is as follows:

> **EH:** The lemma ∀ x_1 ... x_n, s = t should be used as a left to right rewrite rule when the term t in the lemma statement embeds into the term s using first-order embeddings and s is syntactically different from t.

In the simplest cases, the RHS side of a selected equation is a term that occurs in the LHS of the equation. For example, the following equations are identified as left to right rewrite rules for simplification by EH:

```
∀ x, x ∗ 1 = x
∀ x, x ++ [] = x
∀ start len, length (seq start len) = len
∀ a, rev (rev a) = a
∀ a, mirror (mirror a) = a
∀ x, max x 0 = x
∀ x, min x 0 = 0
∀ x, x ∗ 0 = 0
```

In more complex cases, the RHS is not a subterm of the LHS. For example:

```
∀ x y, max x (max y x) = max x y.
∀ a, length (rev a) = length a
∀ f a, length (map f a) = length a
∀ a, num_nodes (mirror a) = num_nodes a
∀ f a, num_nodes (map f a) = num_nodes a
∀ a, sum (rev a) = sum a
∀ a x, list_count (rev a) x = list_count a x
∀ h x a, h ≠ x → list_count (a ++ [h]) x = list_count a x
```

We find the above heuristic identifies many useful simplification rules. For example, as seen above, there exists many simplification rules of the form ∀ x, f (g x) = f x.

Notice that, in cases where the RHS is both a basic term and a subterm of the LHS, the lemma will be selected by both EH and BH. However, BH is restricted to selecting lemmas where the RHS is a basic term which is not the case for EH.

7.11.2.4 Termination

We now justify that exhaustively rewriting a term t using any combination of rules selected by BH and EH must always terminate:

- Let m(s) be the function that sums 1) the number of function symbols that are neither type constructors nor constructors in the term s (e.g. + would be counted but list would not) with 2) the number of variables in s whose type are neither **Set** nor **Type**.

- Let n(s) be the function that returns the number of nodes in the syntax tree for s.

- When a BH rule is used to transform the term t to the term t' by replacing some subterm s in t with the term s', it must hold that m(s) $>$ 0 and m(s') = 0 and therefore m(t) $<$ m(t') must hold. For example, BH rules can only ever be used to eliminate, and never introduce, function symbols like + and variables of type nat so BH rules must always decrease the value of m(t').

- When an EH rule is used to transform the term t to the term t' by replacing some subterm s in t with the term s', it must hold that m(s) $\leq$ m(s') and n(s) $<$ n(s') and therefore m(t) $\leq$ m(t') and n(t) $<$ n(t') must hold. Specifically, an EH rule can only transform s by stripping nodes from its syntax tree and can never introduce new nodes.

- Exhaustive rewriting with any rules selected by BH or EH rules must terminate as the pair (m(t), n(t)) descends lexicographically each time a rule is used to rewrite t.

7.12 Lemma Discovery

As demonstrated by IsaCoSy [Johansson, 2009], conjecturing and caching lemmas about function definitions prior to attempting to prove the theorems we are interested in can improve the proof coverage of rippling-based proof automation. We describe here a limited form of lemma discovery we use that aims to only conjecture a small number of common lemma forms. This is intended to be used whenever a new simply typed function definition is entered. The procedure we use for lemma discovery is as follows:

1. The system is provided with a list of hand-crafted lemma templates that state generic operator laws. We make use of templates for involution, commutativity and associativity laws. For example, the template ∀ (x y:t), f x y = f y x describes commutativity, where f must be instantiated to some binary function f and type inference is used to instantiate t with an appropriate type.

2. After a new simply typed function g is defined by the user, g is used to instantiate all available lemma templates to create a list of terms representing conjectures. For example, after plus is defined, the template given above would be instantiated to create the conjecture ∀ (x y:nat), plus x y = plus y x. In cases where g has arguments of type **Type**, all these arguments are instantiated to the same universally quantified variable T that has type **Type**. For example, consider if f in the template ∀ (x:t), f (f x) = f x was to be instantiated with the following function:

   ```
   rev : ∀ (T:Type), list T → list T
   ```

   The instantiated template would then have the following form:

   ```
   ∀ (T:Type) (x:list T), rev T (rev T x) = x
   ```

3. Any ill-typed conjectures that are generated are discarded. For example, this would happen when the commutativity template is instantiated with a single argument function.

4. Our testing tool (see Chapter 8) is used to identify and discard faulty conjectures.
5. The proof automation is then used to attempt to prove the remaining conjectures. Successful proofs result in lemmas being cached.

We have implemented partial automation for the above, where lemma discovery must be called manually on a function when it is introduced.

7.13 User Hints

In this section, we describe how our proof automation gives feedback on failed proof attempts and explain the feature that allows the user to help the proof automation by providing hints.

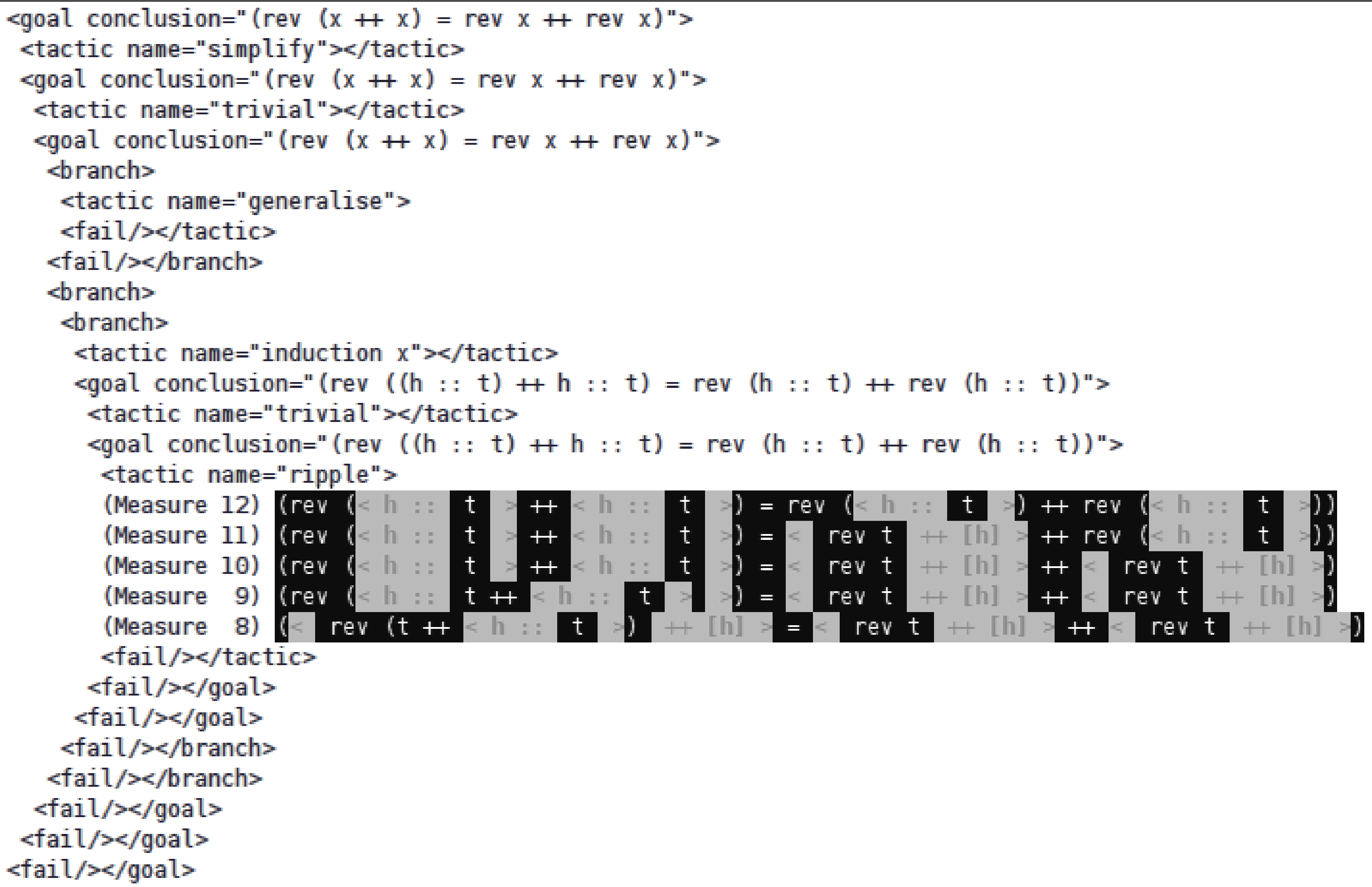
7.13.1 Proof Search Feedback

When a proof attempt fails, the proof automation displays a trace of the failed proof search. The user can utilise the information in the trace to determine ways to provide useful hints to the automation.

A proof trace consists of a tree of goal nodes, where each goal node contains a description of a Coq goal that was seen during proof search. Each goal node is connected with a single directed edge to either a "tactic" node or a "branch" node.

When a goal node `g` is connected to a tactic node, where the latter node is labelled with the name of some Coq tactic `t`, this has the meaning that `t` was invoked on `g` during proof search. Each tactic node is joined with directed edges to the goal nodes that represent the subgoals that were produced when this tactic was invoked during proof search.

When a goal node is connected to a branch node, this represents a point in the search space where backtracking was possible. A branch node is connected with directed edges to one or more tactic nodes. Each of these tactic nodes represents a tactic that was invoked on the goal that was connected to the branch node. Each node in the proof trace is annotated with either "fail" or "success". A goal node and all of its child nodes are annotated with "fail" if the goal this node represents was not discharged during proof search. A goal node and all of its child nodes are annotated with "success" if the goal this node represents was discharged during proof search.



```
<goal conclusion="(rev (x ++ x) = rev x ++ rev x)">
 <tactic name="simplify"></tactic>
 <goal conclusion="(rev (x ++ x) = rev x ++ rev x)">
  <tactic name="trivial"></tactic>
  <goal conclusion="(rev (x ++ x) = rev x ++ rev x)">
   <branch>
    <tactic name="generalise">
    <fail/></tactic>
   <fail/></branch>
   <branch>
    <branch>
     <tactic name="induction x"></tactic>
     <goal conclusion="(rev ((h :: t) ++ h :: t) = rev (h :: t) ++ rev (h :: t))">
      <tactic name="trivial"></tactic>
      <goal conclusion="(rev ((h :: t) ++ h :: t) = rev (h :: t) ++ rev (h :: t))">
       <tactic name="ripple">
       (Measure 12) (rev (< h :: t > ++ < h :: t >) = rev (< h :: t >) ++ rev (< h :: t >))
       (Measure 11) (rev (< h :: t > ++ < h :: t >) = < rev t ++ [h] > ++ rev (< h :: t >))
       (Measure 10) (rev (< h :: t > ++ < h :: t >) = < rev t ++ [h] > ++ < rev t ++ [h] >)
       (Measure  9) (rev (< h :: t ++ < h :: t > >) = < rev t ++ [h] > ++ < rev t ++ [h] >)
       (Measure  8) (< rev (t ++ < h :: t >) ++ [h] > = < rev t ++ [h] > ++ < rev t ++ [h] >)
       <fail/></tactic>
      <fail/></goal>
     <fail/></goal>
    <fail/></branch>
   <fail/></branch>
  <fail/></goal>
 <fail/></goal>
<fail/></goal>
```

Figure 7.1: Screenshot showing a trace of a failed proof attempt.

7.13.1.1 XML Representation

A proof trace of the form described above is displayed by the prover in the form of an XML tree. For example, Figure 7.1 shows the failed proof trace produced when the prover is asked to prove $\forall$ x, rev (x ++ x) = rev x ++ rev x from basic definitions. The indentation of the XML represents the depth of the search. Our intention was that the XML tree would be converted to a more human readable format before being displayed to the user but this was not done due to time constraints. This particular trace shows that the prover failed to simplify or generalise the goal, performed induction on x, failed to fertilise in the rippling proof for the step case and then the proof attempt failed. We explain in the next section how the user can provide a hint to automate this particular proof. We now describe the meaning of the XML tags:

- A goal tag describes the state of the goal at a given stage in the proof attempt. Tags nested within a goal tag describe the attempt to prove that goal.

- A tactic tag gives the name of the high-level tactic that was invoked on the current goal. Each goal tag that appears in sequence after the closing tactic tag represents a subgoal produced by the tactic that was called.

- A <fail/> tag that comes before the tag that closes a tactic, goal or branch block indicates that the last tactic call failed and backtracking occurred.

- Tag sequences of the form <branch>t_1</branch><branch>t_2</branch>... represent choice points in the search space, meaning what is described in t_1 was performed first, backtracking occurred, then what is described in t_2 was performed next and so on. The branch tags in this trace indicate the choice of whether to generalise the goal or not, and the choice of which variable to perform induction on.

- Between the opening and closing tags for a tactic tag, trace information produced by the tactic called is displayed. Of particular interest, the ripple tactic will display the sequence of transformations it makes to the conclusion during the rippling proof search. For each transformation, the current ripple measure and the wave front annotations are shown. To improve readability, wave fronts are coloured differently from the rest of the terms. We find this helps significantly when inspecting rippling proof attempts.

Proof traces are written to the standard output stream so they can be displayed in a terminal window alongside Coq's standard IDE. We initially tried displaying proof traces inside the error feedback window in Coq's IDE, but we found the length and width of typical proof traces were too large for this to be practical.

For future work, it would be useful to present the user with a graphical overview of proof attempts. For example, a so-called hiproof could be presented to explain to the user which subgoals were considered and which tactics were invoked in a proof attempt [Denney et al., 2006, Aspinall et al., 2008].

7.13.2 User Hinting Mechanism

After a failed proof attempt, the user can invoke the `hint(c)` command to give conjecture `c` as a hint, where this command is entered as part of the program script. The prover then tries to prove `c`, where the following scenarios can occur:

- If the prover is successful, `c` and its proof will be cached as a lemma in the same manner as lemmas are cached during proof search. As described in §7.2, cached lemmas of appropriate forms can be automatically utilised by the `ripple`, `trivial` and `simplify` tactics.

  The failed proof attempt from before can then be reattempted, with the hope that the extra lemma can be used productively by the prover to complete the proof this time. Note that the proof is reattempted from scratch when a hint is given and we do not reuse any progress from the previous search.

- If the prover is unable to prove `c`, a proof trace is displayed. In such cases, the user can give a further hint to help produce a proof for `c`.

- The hint may be faulty in that `c` is a nontheorem. In §8.2.2, we describe how we can make use of our testing tool to give useful feedback in the form of a counterexample to help the user refine faulty hints.

7.13.3 Providing Productive Hints

To give useful hints, the user must consider how their hint could be productively used by the various tactics. The following describes the primary ways that useful hints can be given by making use of the information for proof traces:

Wave rule hints: When a proof trace indicates that the ripple tactic failed to fertilise in a goal, the user can conjecture a lemma that can be used as a wave rule to help. The wave front annotations in the proof trace can give the user a strong indication about the shapes of lemmas which would be useful for this.

Generalisation hints: When a proof trace indicates that the generalise tactic did not generalise some goal g appropriately, the user can suggest a generalised form of the goal as a hint. To do this, the user would suggest the conjecture g' as a hint, where g' subsumes g. If the lemma for g' is proven, this lemma can then be used by the trivial tactic to automatically prove g.

Returning to the example in the previous section, the proof trace shows the generalise tactic was unable to suggest a way to generalise the top-level goal. The top-level goal can be stated more generally, as we show with the following hint that the prover is able to automate:

```
hint(∀ x y, rev (x ++ y) = rev y ++ rev x).
```

Simplification hints: The user can gives hints to help the simplify tactic perform more effective simplification. For example, if the user noticed a unprovable subgoal contained the term rev (rev x), the user might be able to help the prover succeed by giving the following hint:

```
hint(∀ x, rev (rev x) = x).
```

The user can then use an extra command for manually adding the lemma proven for this hint to the rewrite rule database used by the simplify tactic. However, this manual step is not required for this particular rule as the prover has heuristics that will automatically use this rule for simplification (see §7.11).

It is worth noting the similarities between the role of the user in providing hints and the role of a proof critic in proof planning [Ireland and Bundy, 1996]. Similarly, a proof critic uses the information gained from a failed proof attempt to suggest a way to patch the failing proof. As such, it is possible that critics could be applied here to reduce the need for manual user hints. However, as proof critics are not guaranteed to succeed in all cases, we feel it is important to provide facilities for manual user hinting.

7.14 A Comparison with IsaPlanner

In this section, we evaluate the utility of our top-level tactic as an inductive proof automation tool. To do this, we evaluate our tactic against a theorem corpus that has been previously used to evaluate the inductive proof automation power of IsaPlanner [Johansson, 2009, §5.5]. This corpus contains 87 theorems concerning arithmetic, lists and binary trees, many of which are taken from Isabelle's standard theorem libraries. The corpus can be found in Appendix C, where we make references to the labelled theorems in this section. The corpus was devised to test IsaPlanner's ability to automate proofs that require case splits to be performed. IsaPlanner was able to automate 47 of these theorems. In this section, we test our prover against the same theorems and compare the results with IsaPlanner.

7.14.1 Experimental Setup

When IsaPlanner was evaluated against the theorem corpus, only basic function definitions (and no extras lemmas) were supplied to IsaPlanner. For a fair comparison, we configured our prover to only use the same definitions.

To perform the experiment, we first had to translate the IsaPlanner theorem corpus and function definitions to Coq. This translation was mostly straightforward (see Appendix A.4 for the definitions used) except for the following design choices:

- The function last x (which returns the last item in the list x) was defined in Isabelle as a partial function. Specifically, the case in which x is empty is ignored. As Coq does not allow partial functions to be defined, we used the approach of implementing this as the function last x d which returns the last item in list x when x is nonempty and returns a default result d when x is empty.

- In Isabelle, theorem 11 (see Appendix C) has the form $(\mathsf{max}\ a\ b = a) = (b \leq a)$. A literal translation of this statement to Coq is a nontheorem. A natural Coq interpretation of this statement that is a theorem is $(\mathsf{max}\ a\ b = a) \leftrightarrow (b \leq a)$, where the outermost = operator has been replaced with $\leftrightarrow$. A similar interpretation was required to translate theorems 12, 15 and 16 from Isabelle.

7.14.2 Results

Of the 87 theorems, our prover was able to automate 45 (52 %) of these. IsaPlanner was able to automate 47 (54%) theorems. Of the theorems IsaPlanner could automate (those labelled 1 to 47) we could automate 39. From the remaining theorems IsaPlanner could not automate (those labelled 48 to 87), we could automate 6 of these. See Appendix C for the detailed results of which theorems could be automated by our prover.

7.14.3 Analysis

As our rippling tactic relies on the same ripple measure and implements the same technique for reasoning about case splits, we would expect that our system could automate many of the rippling proofs that IsaPlanner can automate. The results of the experiment show that this is the case with only a few exceptions. We now give an analysis of the results of our prover compared to IsaPlanner. We first consider the 8 theorems that our prover failed to automate that IsaPlanner was able to prove:

- Theorems 6, 11, 12, 15 and 16 are not equality statements. We currently only support rippling proofs where the conclusion and the given are equality statements so the inductive proof attempts for these theorems fail. Specifically, the weak fertilisation step of our prover will only work when the given is an equation. These theorems succeed in IsaPlanner as the Isabelle version of the theorems have the form P = Q (see §7.14.1), which IsaPlanner's rippling tactic can support.

- Theorem 31 has the form member x l $\rightarrow$member x (l @ t). Notice that, after performing universal introduction, the goal for this theorem will contain an assumption that embeds into the conclusion. Our top-level tactic therefore invokes rippling on this goal (see §7.1). The proof attempt fails because appropriate lemmas are not available to complete the rippling proof. Theorem 32 fails for the same reason.

  IsaPlanner only ever invokes rippling in step case goals and never on top-level goals. IsaPlanner automates both of these theorems by first performing induction on the top-level goal and then using rippling in the step cases.

  The behaviour of our system here was somewhat surprising as we usually only expect our rippling tactic to be invoked in recursive call proof obligations and in the step cases of inductive proofs. Our prover can automate both of these proofs if it is forced to perform induction on the top-level goal instead of rippling.

To remedy the above problem, we could consider hard-wiring our top-level tactic to only use rippling in recursive call proof obligations and in the step cases of inductive proofs. However, while verifying a quicksort program, we came across a surprising situation where rippling was applicable in a useful way after a top-level goal was simplified (see §9.4.3). We would need to consider more examples to determine the best general approach for when rippling should and should not be invoked when embeddings are found outside of step case goals.

- When proving theorem 43, IsaPlanner performs a case split during a rippling proof where one of the subgoals produced contains an assumption of the form x $\neq$ x. When IsaPlanner performs case splits, it automatically discharges any subgoals generated that contain contradictions of this form [Johansson, 2009]. As we do not check for contradictions when performing case splits during rippling proofs, our rippling tactic fails to automate this subgoal. This could be fixed by performing a call to our trivial tactic when case splits are performed.

We now consider the 6 theorems we are able to automate that IsaPlanner could not:

- Theorem 76 has the form butlast (xs ++ ys) = **match** ys **with** [] $\Rightarrow$ $\ldots$. Our prover proceeds by performing induction on the goal for this theorem. Unexpectedly, the rippling tactic is not invoked in the step case. The reason for this is that our algorithm for generating rippling annotations cannot annotate conclusions that contain **match** constructs (see §7.8.2). As the rippling tactic reports that there are no assumptions that embed, the top-level tactic performs simplification on the goal (where the inductive hypothesis is never used) and, after another similar inductive proof, eventually discharges the goal. Theorem 80 is proven in a similar fashion. IsaPlanner fails to automate these theorems by attempting rippling in the step case goals. As the prove our prover found did not make use of the inductive hypotheses in the step case goals, this indicates that case analysis would be more suitable for automating these theorems than induction.

- Whenever our prover performs induction, it first modifies the conclusion of the goal to avoid introducing implications into inductive hypotheses (see §7.6). For example, when proving theorem 87, a simple inductive proof is performed on the modified goal $\mathsf{n} \neq \mathsf{h} \vdash \mathsf{count\ n\ (x\ ++\ (h\ ::\ []))}$ as opposed to the initial goal $\vdash \mathsf{n} \neq \mathsf{h} \rightarrow \mathsf{count\ n\ (x\ ++\ (h\ ::\ []))}$. Theorem 69 is automated in a similar manner. In IsaPlanner, induction is performed in such a way that the inductive hypotheses in the step cases for these examples contain implications. IsaPlanner is known to fail in these situations as its rippling tactic lacks support for step cases of this form [Johansson, 2009, §5.6].

- Theorems 74 and 84 are automated by our prover using simplification followed by induction. For example, for theorem 74, the initial goal is:

  $\mathsf{ys0 = [\,] \rightarrow last\ (xs\ ++\ ys0)\ default = last\ xs\ default}$

  This goal is simplified to $\mathsf{last\ (xs\ ++\ [])\ default = last\ xs\ default}$ and the inductive proof is straightforward. IsaPlanner does not simplify top-level goals before performing induction and fails to automate these theorems using this strategy.

For the remaining theorems that cannot be automated by either system, our prover fails in these proof attempts for the same primary reasons that IsaPlanner does [Johansson, 2009, §5.6]. Specifically, both of our systems lack support for using induction principles other than structural induction and cannot perform rippling proofs that require piecewise fertilisation [Armando et al., 1999].

7.14.4 Summary

In this experiment, we observed that IsaPlanner and our prover provide similar proof coverage for the theorem corpus that was considered. The results did however highlight the difference in the automation approach used by the two systems. Specifically, our prover always attempts simplification on top-level goals and performs rippling if applicable. In contrast, IsaPlanner never simplifies top-level goals and will only perform rippling in step case proofs. We identified cases where simplifying the top-level goal would improve the proof coverage of IsaPlanner and observed the unexpected behaviour that allowing rippling to be invoked in top-level goals can block proofs in some situations.

It should be noted that the theorem corpus used in this case contained theorems where rippling and simplification were rarely applicable to the top-level goals. In §9.8, we compare the proof coverage of these two systems against proof obligations generated from dependently typed programs, where our prover is found to perform significantly better.

7.15 Conclusions

We have described tactics designed to automate the proof patterns that arise when programming with dependent types. This automation uses heuristics for simplifying and generalising goals, and employs the rippling heuristic for guiding induction-like proofs. Of note, several design choices were made so that more general, and thus reusable lemmas would be cached during proof attempts. In particular, we use a liberal generalisation tactic and the delayed generalisation algorithm is used to identify and remove irrelevant subformulae from cached lemmas. We further improve the reusability of cached lemmas by adding heuristics to identify those which are appropriate for use by the simplification tactic. We then described the proof traces that are produced when proof search fails and explained how the user can provide hints that can help the prover overcome failures. In the next chapter, we introduce our testing tool and explain how this is integrated into our framework to provide further support.

Chapter 8

Supporting Dependently Typed Programming with Testing

In this chapter, we explain the role of testing in our framework for supporting dependently typed programming. In the next two sections, we describe how testing is utilised in the following ways:

1. When a dependently typed function generates an unprovable proof obligation, testing is used to identify this and an error message is shown to the user (see §8.1). The error message shows a counterexample that demonstrates why the identified proof obligation is unprovable. When no counterexample can be found to a proof obligation and the automation fails to find a proof, the user is presented with a trace of the failed proof attempt (the latter feedback was described previously in §7.13).

2. Testing is used by the proof automation to prune overgeneralisations from the search space (see §8.2.1).

3. When the user supplies a non-theorem as a hint to the prover, testing is used to give feedback to help the user fix their faulty hint (see §8.2.2).

4. When the user is performing a manual proof, testing can be used as a tool during the proof attempt to identify unprovable goals (see §8.2.3).

We describe a testing tool that we have designed and developed for Coq for the above purposes in §8.3. In the meantime, it suffices to know that this tool uses a QuickCheck-like approach [Claessen and Hughes, 2000] for finding counterexamples to universally quantified conjectures.

8.1 Providing Error Feedback with Testing

In §4.3, we described how identifying and fixing errors in dependently typed programs can be challenging. Recall that the kinds of errors we are interested in are those indicated by unprovable proof obligations. We now describe how we can apply testing to identify such errors and provide useful feedback. Before explaining the implementation details of our procedure for providing error feedback, we give an example of a program that contains an error and show the style of feedback that we want to provide. For this example, we want to define the function intersperse x y where intersperse returns the list x with the items in list y inserted after every item. For instance, intersperse [1; 2; 3] [4; 5] would return [1; 4; 5; 2; 4; 5; 3; 4; 5]. We also want to verify the length of the output list from intersperse is correct using subset types. The following faulty definition almost achieves this task but contains an error:

```
Program Fixpoint intersperse (x : list nat) (y : list nat) :
 {r : list nat | length r = (length x) * (length y)} :=
  match x with
  | [] ⇒ []
  | h::t ⇒ [h] ++ y ++ (intersperse t y)
  end.
```

The body of the function has the expected behaviour but the output type is faulty. This faulty typing leads to the recursive call proof obligation being unprovable. The mistake made is that the output type should actually be the following (notice the extra S term):

```
{r : list nat | length r = (length x) * S (length y)}
```

The unprovable recursive call proof obligation produced by the faulty function has the following form after destructuring the recursive call and substituting pattern matching equations (as we detail in §8.1.1, these steps form part of the testing procedure):

```
y : list nat
h : nat
t : list nat
intersperse_s : list nat
intersperse_p : length intersperse_s = length t * length y
─────────────────────────────────────────────────────────────
length ([h] ++ y ++ intersperse_s) = length (h :: t) * length y
```

We find the above unprovable proof obligation typical, in that it is non-obvious from casual inspection that the proof obligation is unprovable. Furthermore, even when we know the above is unprovable, identifying where the fault lies is challenging.

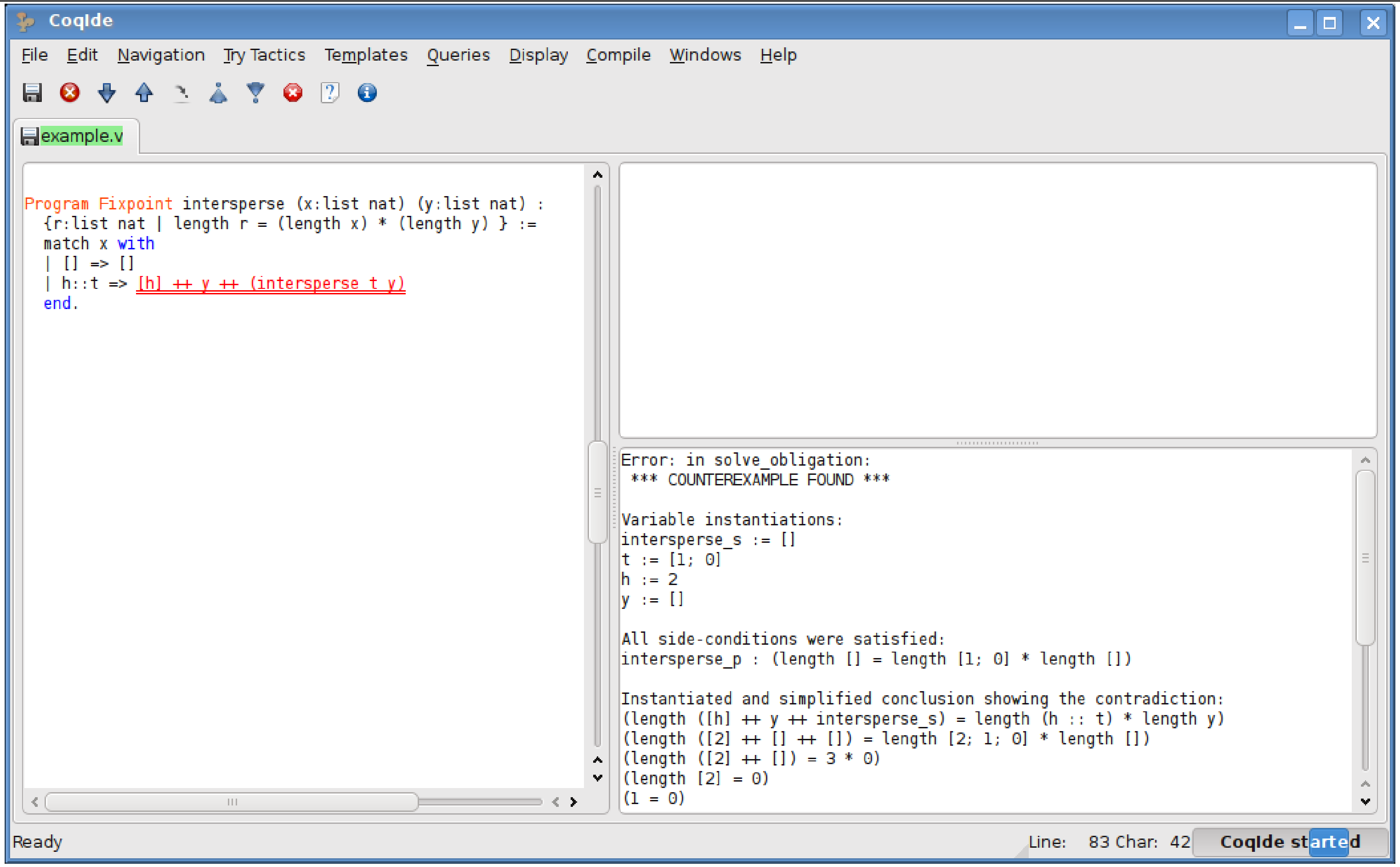
We propose the application of QuickCheck style testing on proof goals for identifying unprovable proof obligations and the use of counterexamples found during testing for providing helpful error feedback. To do this, whenever a proof obligation is generated, the testing tool we have designed is automatically invoked on each proof obligation. For the above proof obligation, the testing tool will quickly find a counterexample. When this happens, the term that produced the unprovable proof obligation is underlined in Coq's IDE [1] and an error message containing the counterexample found with an evaluation trace is displayed to the user (see Figure 8.1). The error message displayed in the case of this example is as follows:

```
*** COUNTEREXAMPLE FOUND ***

Variable instantiations:
y := [], h := 1, t := [], intersperse_s := []

All side-conditions were satisfied:
intersperse_p : length [] = length [] * length []

Instantiated and simplified conclusion showing the contradiction:
length ([h] ++ y ++ intersperse_s) = length (h :: t) * length y
length ([1] ++ [] ++ []) = length [1] * length []
length ([1] ++ []) = 1 * 0
length [1] = 0
1 = 0
```

The error message contains the variable instantiations for the counterexample and shows, with a step-by-step trace, how the conclusion evaluates to a contradiction. This information is intended to be used by the user to isolate the cause of the error and give hints to what changes need to be made.

In the above, we can see that when h is 1 and t, y and intersperse_s are empty, this leads to a contradiction (i.e. when the input to intersperse is [1] []). Notice that the LHS of the conclusion concerns the term from the implementation of intersperse that produced the proof obligation. The RHS of the conclusion is supposed to capture the length of the list we expect from a valid implementation of intersperse in this case.

[1]Thanks to Matthieu Sozeau for making the modification needed to his Program tactic to allow for this behaviour.


CoqIde
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
example.v
Ready
Line: 83 Char: 42
CoqIde started

```
Program Fixpoint intersperse (x:list nat) (y:list nat) :
  {r:list nat | length r = (length x) * (length y) } :=
  match x with
  | [] => []
  | h::t => [h] ++ y ++ (intersperse t y)
  end.
```

```
Error: in solve_obligation:
 *** COUNTEREXAMPLE FOUND ***

Variable instantiations:
intersperse_s := []
t := [1; 0]
h := 2
y := []

All side-conditions were satisfied:
intersperse_p : (length [] = length [1; 0] * length [])

Instantiated and simplified conclusion showing the contradiction:
(length ([h] ++ y ++ intersperse_s) = length (h :: t) * length y)
(length ([2] ++ [] ++ []) = length [2; 1; 0] * length [])
(length ([2] ++ []) = 3 * 0)
(length [2] = 0)
(1 = 0)
```

Figure 8.1: A screenshot that demonstrates what happens in Coq's IDE when an unprovable proof obligation is identified. The term in the program script in the left pane is underlined to indicate this term produced the unprovable proof obligation. The bottom right pane shows the counterexample-based error message.

By considering that a valid implementation of intersperse should return a list of length 1 for the given variable instantiations (i.e. because y is empty, the output list should have the same length as list x), we can reason that the LHS is correct but the RHS is incorrect. The latter indicates that the error is caused by a faulty output type. The evaluation trace, which shows how the value for the RHS of the conclusion was calculated for this counterexample, suggests that adding an S around the length y term would fix the problem.

Furthermore, the evaluation trace is useful when a function that appears in the proof obligation is faulty. For example, if length had been wrongly defined to always return 0, this would become obvious from looking at the steps in the evaluation trace.

8.1.1 Error Feedback Procedure

We now describe the procedure used for generating the error feedback shown above. Whenever a dependently typed program generates proof obligations, we apply the following procedure to each proof obligation before our proof automation is invoked:

1. Exhaustive universal introduction is performed on the proof goal and pattern matching equations are substituted.

2. All subset type terms in the conclusion and subset type assumptions are destructured. This is currently required for our testing tool to work.

   As it is important for the user to be able to trace the origin of the terms produced here back to their program, we must employ some form of origin tracking [van Deursen et al., 1993]. As we only perform minimal modifications to a goal before testing occurs, we use the following basic procedure for this: when destructuring the subset type term returned by a call to function f, the two assumptions produced are given labels of the form f_s and f_p for the computational term and the propositional term respectively so that the user can determine their origin from the labels.

3. The testing tool is then invoked on the proof goal, resulting in one of the following:

   - If the goal is falsified, we display the counterexample found with an evaluation trace in the form of an error message. We explain how to create concise evaluation traces in §8.1.2. Additionally, the term that produced the unprovable proof obligation is indicated to the user.

- If the goal cannot be falsified, the proof automation is invoked on the original unmodified goal to attempt to discharge the proof obligation. The modifications to the goal must be undone as these can interfere with the `recursive_call` proof pattern (see §6.5).

  Note that finding a counterexample to the modified goal in step 3 means that a counterexample exists for the top-level goal as performing universal introduction and destructing subset types can never turn a provable goal into an unprovable goal or vice versa.

8.1.2 Concise Counterexample Evaluation Traces

For conciseness, we use a procedure that generates compact yet easy to follow evaluation traces of counterexamples when showing error messages. Notice in the evaluation trace from the previous section how incremental simplification is performed on both the LHS and RHS of the equation on each line. The underlying motivation was to replicate how an algebraic equation is simplified over several lines in pen-and-paper proofs. Typically, a handful of simplifications are made on each new line in a way that keeps the number of lines short yet maintains the checkability of the steps that led to the final result.

To make the simplifications for one line of the evaluation trace, the procedure used performs a postorder traversal (i.e. the innermost subterms are considered first) of the syntax tree of the current conclusion term. For each node `n` in the tree, we simplify `n` by performing computations (i.e. using Coq's `simpl` tactic [Bertot and Castéran, 2004]) only if none of the child nodes of `n` have been modified so far. The above procedure is implemented in Coq as an OCaml function that, when supplied with a Coq term, displays an evaluation trace. Specifically, this procedure is not implemented as a tactic as no proof is being constructed.

For example, consider if the conclusion was the following term:

```
1 + (2 + 3) = 4 + 5
```

In one traversal, the first subterms that are simplified are `2 + 3` and `4 + 5`. This simplifies the conclusion to `1 + 5 = 9`. After this, no more terms are simplified for this line in the evaluation trace. In particular, the terms `1 + 5` and `1 + 5 = 9` are not simplified because these terms are composed of subterms that have been simplified already. For the next line of the evaluation trace, the conclusion is simplified to 6 = 9. The evaluation trace is then finished as there are no more terms to simplify.

8.1.3 Weak Specifications and Counterexamples

We note that weakly specified programs (see §3.4.1) can be somewhat problematic when generating easy-to-understand error messages. For example, reconsider the unprovable proof obligation from the weakly specified interperse program, where only the length property of the output list was captured:

```
y : list nat
h : nat
t : list nat
intersperse_s : list nat
intersperse_p : length intersperse_s = length t * length y
─────────────────────────────────────────────────────────────
length ([h] ++ y ++ intersperse_s) = length (h :: t) * length y
```

Given y := [1] and t := [2], a valid instantiation for the recursive call result intersperse_s that satisfies the constraint given by intersperse_p is [3]. However, the computational part of the definition we gave for interperse could never return such a result from the recursive call given those instantiations for y and t.

The issue here is that the terms generated for the counterexample only have to satisfy the constraints given by the weak specification of the intersperse function i.e. as specified by the intersperse_p assumption. Specifically, the intersperse_p assumption constrains the length of the lists generated but the contents of the length are unconstrained. The user must keep this in mind when interpreting the error messages. This issue is not present with strongly specified functions as the valid term instantiations will be precisely constrained by the assumptions.

We note that the error messages produced when working with weak specifications could be improved if the testing procedure generated counterexamples by making use of the computational content of the function the user was trying to define. However, the latter information is currently not accessible in the proof obligations Russell produces.

8.2 Testing and Proving

In this section, we explain how testing is integrated with the proof automation described in the previous chapter to provide extra support for the user.

8.2.1 Testing as Part of Proof Automation

During proof search, we use testing to prune unprovable proof goals from the search space. Nontheorems can result from the overgeneralisations produced by the generalise tactic (see §7.5). This use of testing during proof search has the following benefits:

Efficiency: Attempting to prove nontheorems can have a significant performance cost as sometimes large search trees must be exhausted for the proof automation to terminate. For example, a nontheorem that contains many variables can result in a large proof search tree as each variable could be considered a candidate for induction by the induction tactic.

Concise Search Traces: When proof search fails, determining what hints might be useful to help the prover can involve inspecting a trace of the failed proof attempt (see §7.13). Pruning fruitless paths from the search tree makes this task easier as the trace will be more concise.

8.2.2 Feedback for Faulty Hints with Testing

In §7.13, we describe the feature of our proof automation where the user can aid failed proof searches by supplying appropriate lemma hints. To improve the usability of our framework, we employ testing to identify cases where the user supplies a nontheorem as a lemma hint.

When the user supplies a conjecture as a lemma hint, we use testing, before invoking the proof automation, to attempt to falsify the conjecture. If a counterexample is found, this is shown to the user in the form of an error message (in the same way as shown in §8.1). The use of testing to identify nontheorems supplied as lemma hints has the following benefits:

Nontheorem detection: If the user is only told that the prover was unable to prove the conjecture, the user may make further attempts to prove the conjecture themselves. When the conjecture is identified as a nontheorem, this will prevent the user from wasting time attempting a manual proof.

Refinement help: Counterexamples can aid the user in refining nontheorems into theorem statements. For example, an error message might suggest to the user what side-conditions must be added to refine their nontheorem into a theorem. In future work, automation could be provided here by using counterexamples to guide the refinement of a non-theorem into a theorem [Colton and Pease, 2005].

8.2.3 Supporting Manual Proofs with Testing

For situations where the user decides to construct a proof manually, we have packaged our testing tool as a tactic so that it can be invoked to look for counterexamples to the current goal. Testing is useful for identifying intially unprovable conjectures and identifying proof steps that change a provable goal to an unprovable goal. Additionally, any counterexamples found can help the user refine unprovable conjectures to theorem statements and help explain why a certain proof step was unsafe.

8.3 Design of a Testing Tool

In the previous sections, we described the various ways we can use testing to improve the support given by our framework. In this section, we describe the design of a QuickCheck-like [Claessen and Hughes, 2000] testing tool, that we have implemented in Coq, that can be used for the purposes we have mentioned. As with the testing tool for Agda, testing in our tool occurs within the framework of the goals being tested [Qiao Haiyan, 2003].

8.3.1 Requirements

The following describes what requirements the testing tool we have designed is intended to meet:

Coverage: When used for providing error feedback, the testing tool should be able to identify a significant number of unprovable proof obligations to be useful. If the user-supplied program contains an error which is not identified by testing, the usability penalty can be high as the user can be unsure whether the proof obligation is provable or not. Likewise, we would expect similar reliability when testing is used to identify faulty lemma hints.

Easy to interpret counterexamples: As the user is expected to inspect the counterexamples found when these are shown as error messages, we want the counterexamples to be easy to interpret. It is generally agreed that inspecting a minimal counterexample is easier than inspecting a large and complex one.

Efficiency: Searching for counterexamples needs to be reasonably efficient. For example, we would expect a user to be frustrated if they had to wait more than a few seconds to test the proof obligation generated from their program. Additionally, when testing for unprovable goals during proof search, this only saves time if testing takes less time than it does to wait for failing proof attempts to finish.

8.3.2 Counterexample Generation

We start by giving a high-level overview of our testing tool. To find counterexamples to Coq goals, we use the generate-and-test approach used by QuickCheck-like tools [Claessen and Hughes, 2000]. In contrast to, for example, the testing tool available for Isabelle [Nipkow, 2004], testing is performed within the same framework of the goals being tested. We discuss the merits of this approach later in §8.3.7.

Our testing tool is first supplied with a Coq goal of the following form:

$(H_1 : T_1) \quad \ldots \quad (H_n : T_n) \quad \vdash P$

Firstly, to test a goal, we must have a method to give a concrete instantiation to each variable H : T in the goal when T:**Set** or when T:**Type**. We explain term generation in §8.3.5, where our generator can randomly generate terms for ML-like types (i.e. but not dependent types).

Secondly, P and each assumption H : T where T : **Prop** must be *testable* after the above variable instantiations have been made. We explain how instantiated propositions are tested in §8.3.3. For example, we can provide support for testing propositions of the form s = t, where s and t are instantiated to ground terms, but not for propositions that include quantifiers. The latter limitation is imposed as otherwise we would need some fast procedure for verifying statements of the form (∀ (n:nat), P x) and (∃ (n:nat), Q x) (for some proposition P and Q) hold during testing, where this is problematic for infinite types like nat using the basic QuickCheck approach.

The procedure to search for a counterexample is as follows:

1. For each assumption A : R, A is added to the set
   (a) T when R is the term **Type** or **Set**. These represent type variables that need to be instantiated.
   (b) V when R:**Set** or R:**Type**. These represent variables for which we have to generate concrete terms.
   (c) S when R:**Prop**. These represent side-conditions that need to be tested after instantiations are made.

For example, members of the sets T, V and S might be the terms A : **Type**, x : list A and p : length x = 1 respectively. Testing fails if an assumption does not match any of the patterns given. For instance, a goal with assumption P : **Prop** cannot be tested. However, for the examples considered in this thesis, assumptions of this particular form are unlikely to arise in practice.

2. For each assumption A in T, A is replaced in the conclusion and all assumptions in S and V by a concrete data type. We explain this step in §8.3.4.

3. For each assumption x : R in V, a random term t : R is generated and x is replaced with t in all assumptions in S and the goal conclusion. We explain term generation in §8.3.5.

4. The conclusion and all members of S are then simplified by computation (i.e. in the same way Coq's simpl tactic operates on terms [Bertot and Castéran, 2004]).

5. If all properties in S are true and the conclusion is false when tested, a counterexample has been found. We describe how tests are performed in §8.3.3.

6. The search for a counterexample can be continued by repeating the above from step 2. We stop searching after a user-defined number of attempts is reached where we use 100 as the default.

We label a generated example as "vacuous" when testing shows that a member of S was false when tested. Like QuickCheck, we report the percentage of vacuous examples generated to the user. We now describe the details of some of the above steps.

8.3.3 Testing Propositions

To test a Coq proposition P, we have implemented a function called test that either returns a boolean result when P is testable or fails with an error when P is determined to be untestable. The test function is defined recursively as follows to test proposition P (note that the Coq operators $\wedge$, $\vee$, $\rightarrow$, $\sim$, $<$ and $>$ are represented as inductive predicates):

1. test True returns true.

2. test False returns false.

3. test (P ∧ Q) returns (test P) && (test Q), where && is the boolean "and" operator. Likewise, we can give a similar definition to support the operators ∨, → and ∼.

4. Given n and m have type nat and are both ground terms: test (n < m) returns the result of lt_bool n m, where lt_bool is a function that returns a boolean result that decides if n is less than m. Likewise, we can give a similar definition to support the > operator.

5. Given s and t are ground terms: test (s = t) returns true when s and t are convertible and false when they are not convertible.

6. If the term being tested t does not match any of the above forms, t is untestable. For example, the propositions ∀ x, ..., ∃ x, ... and, given x:nat, x + 1 = 1 are untestable.

Our testing tool fails with a warning when an attempt is made to test an untestable term. In further work, we would use a testing procedure that could be extended to work with user-defined inductive predicates. As we tend to use functions as opposed to inductive predicates to represent program properties in this thesis (see §3.4.4), we find the testing procedure above sufficient for our purposes.

8.3.4 Instantiating Type Variables

Some goals require that type variables be instantiated before they are tested. For example, consider the following goal:

```
A:Type
x:list A
____________________
x = x ++ []
```

To generate x and test the conclusion, we must first instantiate A to some concrete data type. To do this, we currently require the user to use a command prior to testing that tells the testing tool what type a named variable should always be replaced with. For example, when testing a goal such as the above, we can instruct the tool to replace occurrences of any variable in a goal with the name A (which must have type **Type** or **Set**) to nat. Testing will fail with a warning if there are type variables that could not be instantiated. It would be useful, and also simple, to modify the testing tool so that when an instantiation for a type variable is not specified by the user, some appropriate random instantiation is chosen instead from the types available.

8.3.5 Random Term Generation

To generate random terms for use in our testing procedure, we have implemented a random term generator for ML-like simply typed terms e.g. types such as nat, list nat and btree. To generate a random term t that is a member of the inductive type T, we perform the following procedure, where s is a natural number variable supplied to limit the size of the term generated:

1. If T has no base case constructors or T is not an inductively defined type, term generation fails (e.g. we cannot generate terms of type **Prop** or list **Prop**).

2. If s equals 0, we randomly choose a base case constructor c for type T. Otherwise, we randomly choose any constructor c for T.

3. A term is generated for each of constructor c's arguments by repeating the term generation process for each argument type with s set to half of its current value. These generated terms are then used as arguments to constructor c to construct term t.

The use of the variable s guarantees termination and gives some control over the size of the term generated. Custom generators for recursive types in QuickCheck typically use a size parameter for the same purposes.

We do not currently provide facilities for writing custom term generators or support random generation of dependently typed terms such as vect. The testing tool will fail with a warning if it needs to generate a term that has a type that is not supported.

For future work, it would desirable to adopt the feature from Agda's testing tool that allows custom generators to be written for inductive families [Qiao Haiyan, 2003].

8.3.6 Generating Small Counterexamples

We use a simple mechanism, which is also used in QuickCheck [Claessen and Hughes, 2000], to increase the likelihood of generating counterexamples that are close to being minimal. When testing the goal, we set the size parameter of the term generator to match the number of tests that have been performed on the goal so far. For instance, when generating test data to test the current goal for the fifth time, the size parameter for the term generator is set to five. This means that small terms are generated for the initial examples and gradually larger terms are used as more tests are performed. We find this approach is generally effective at finding small, and thus more readable, counterexamples for many goals.

Additionally, this approach makes it easier to find counterexamples in goals with side-conditions (recall that we have not implemented facilities for custom term generators yet). For example, consider the following goal:

```
x : list nat
y : list nat
P : length x = length y
____________________________
 ...
```

If we allow for large random terms to be generated for x and y, it is highly unlikely P will hold for these values. When we limit the term generator size parameter to a small value, we are much more likely to find a pair of lists that satisfy P.

8.3.6.1 Testing and Higher-Order Functions

To test proof goals that contain variables that represent higher-order functions, some mechanism is required for replacing these variables with appropriate concrete functions. The testing tools for Agda [Qiao Haiyan, 2003] and PVS [Owre, 2006], as well as Gast [Koopman et al., 2002], Smallcheck [Runciman et al., 2008] and QuickCheck [Claessen and Hughes, 2000], each have different approaches for dealing with this problem. In each case, there is support for generating random functions for use in testing. For our testing tool, we use a simple solution to provide some support for testing with higher-order functions:

1. The user first supplies the testing tool with a list of functions that should be used when testing goals. For example, the user might supply the functions S : nat → nat and eq_nat_dec : ∀ (x y:nat), {x = y}+{x ≠ y}.

2. To test a goal containing some variable f : T, where T is a function type, f is randomly replaced by a user supplied function that has type T. Testing fails if no such replacement is available.

For example, consider the following unprovable goal:

```
A : Type
B : Type
f : A → B
x : list A
y : list A
─────────────────────────────────────────────────
rev (map f (x ++ y)) = rev (map f x) ++ rev (map f y)
```

The conclusion contains an easily made mistake, where the x and y variables on the RHS of the equation appear in the wrong order. Given that the testing tool has been supplied with the successor function S, the following readable counterexample is easily found by instantiating A to nat and then instantiating f to S:

```
Variable instantiations:
A := nat, B := nat, y := [1], x := [0], f := S

Conclusion:
(rev (map f (x ++ y)) = rev (map f x) ++ rev (map f y))
(rev (map S ([0]++[1])) = rev (map S [0]) ++ rev (map S [1]))
(rev (map S [0; 1]) = rev [1] ++ rev [2])
(rev [1; 2] = [1] ++ [2])
([2; 1] = [1; 2])
```

Despite the obvious limitation of failure occurring when functions of the appropriate types are not available during testing, this simple approach is useful in practice. When counterexamples are generated, replacement with user generated functions arguably increases the readability of the counterexamples compared to replacement with randomly generated functions. With randomly generated functions, the user will be presented with functions that perform unfamiliar computations, which will make understanding the counterexample challenging. With user supplied functions, the user will be familiar with the names and computations of the functions used.

For future work, it would be useful to consider how user-defined functions could be randomly combined to generate new functions that could be used by the testing tool. Compared to only using user-defined functions, this would give access to a larger pool of functions when testing goals. Moreover, compared to functions that are generated completely at random, functions created by combining user-defined functions are likely to be easier to interpret by users when used in error messages.

8.3.7 Testing within Coq

Testing occurs within the framework of Coq, where the term generator generates Coq terms and testing involves simplifying Coq terms by computation. This approach is different to, for example, the testing tool for Isabelle, where Isabelle specifications are first translated to an ML representation and testing takes place in ML for efficiency [Nipkow, 2004]. We find our approach has the following advantages:

- Displaying the counterexample message and the trace with the same notation, formatting and labels used by Coq is trivial. This makes it much easier to generate readable counterexample descriptions and traces.

- We have the option of writing term generators in the same language as the programs we want to test. This approach has been demonstrated in Agda's testing tool [Qiao Haiyan, 2003].

- We can avoid the issue of unsound counterexamples being found when converting between representations e.g. when arbitrary arithmetic precision is used by only one language.

  However, this issue is harder to avoid if, for example, the goal of constructing a verified Coq program was to extract this program to ML and then use machine integers in place of nat terms in the ML code. In this scenario, we would want our testing tool to identify properties that hold in Coq but do not hold in the ML code. One approach to increase confidence that the properties of the verified code hold for the extracted code is to perform testing on the extracted code. This approach is used in the Focal programming language [Carlier and Dubois, 2008].

However, one issue we did encounter with our approach is that evaluating Coq terms can be slow when the operations being used produce large Coq terms. For example, to evaluate the result of $200 * 200$ using Peano arithmetic in Coq takes 0.35s (using an Intel E5200 CPU and 4Gb of RAM). This problem can be worked around by configuring the term generator to only generate small terms when conducting tests (see §8.3.6). With this approach it only takes 0.04 seconds on average for our testing tool to run 100 tests on each proof obligation generated from our cases study programs from Chapter 9.

8.4 Related Work

We now mention some related testing tools available for programming languages and proof assistants. One of the most well-known testing tools is QuickCheck, which is built for testing Haskell programs [Claessen and Hughes, 2000]. Our Coq testing tool is based on the same approach QuickCheck uses for testing goals (see §2.9.1). In particular, both systems generate random test data when searching for counterexamples. However, unlike our system, QuickCheck provides facilities for writing custom generators so that test data can be generated more efficiently.

The approach used by SmallCheck for generating test data is to exhaustively search, up to some size limit, all possible term instantiations [Runciman et al., 2008]. This has the benefit of finding the smallest possible counterexample, which is useful for generating readable feedback. The Gast tool for the language Clean employs a combination of the systematic checking, that SmallCheck uses, and random testing [Koopman et al., 2002]. A QuickCheck-like testing also exists for the Focal environment [Carlier and Dubois, 2008]. This tool has similarities to ours in that both generate test data by randomly selecting constructors from the type we wish to construct [Carlier and Dubois, 2008].

There are also QuickCheck inspired tools available for many proof assistants. Most closely related to our work is the testing tool for Agda [Qiao Haiyan, 2003] as these both function in a dependently typed setting. As with our work, this tool performs testing within the same framework that the goals being tested are represented in. Custom generators can be written for Agda's tool, where the generator functions are implemented as Agda functions. A practical benefit of this is that a generator can be proved to be a surjective function within Agda [Qiao Haiyan, 2003]. Agda also offers support for generating dependently typed terms as well as random functions.

The QuickCheck-like tool for Isabelle takes a different approach to testing by first translating the goals being tested to ML [Nipkow, 2004]. This is in contrast to our tool, where testing takes place within the same environment as the goal we want to test. The Isabelle tool supports testing for goals that include inductive datatypes as well as inductive predicates, where we do not support the latter. A QuickCheck tool has also been developed for PVS which, similarly to our work, can be used to test goals that include subset types [Owre, 2006].

Another approach to testing is to translate the goal we want to test to propositional logic and then employ a SAT solver. This technique is used to test first-order goals in MACE [McCune, 2001] and higher-order goals in both Refute [Weber, 2008] and Nitpick [Blanchette and Nipkow, 2009].

8.5 Conclusions

In this chapter, we described the testing tool component of our framework and how this is integrated with our proof automation to provide further support for programming with dependent types. The testing component employs a QuickCheck-like approach to find counterexamples to unprovable proof obligations, where we presented an implementation of such a tool for the Coq proof assistant. One purpose of testing in our framework is to provide error feedback to the user in the form of counterexample descriptions for unprovable proof obligations. We explained how testing is used to give feedback to faulty user hints as well as to identify overgeneralisations made by our prover during proof search. The testing tool can also be manually employed to help during interactive proofs. In the next chapter, we make use of our testing tool as part of our complete prototype to conduct case studies that examine what support our framework can give when programming with dependent types.

Chapter 9

Case Studies

We have now described the design of our framework for supporting dependently typed programming. In this chapter, we evaluate our claim that this framework can make development significantly easier. To do this, we discuss the results of several case studies where the framework was used to provide help for writing dependently typed programs. The case studies are used to illustrate the strengths and weaknesses of our framework and compare the support it gives to currently available programming environments.

9.1 Research Questions

We first consider the research questions that we would like our case studies to answer. Regarding the support provided for developing dependently typed programs, we would like to answer the following questions:

- Which data type representations, program property representations and levels of type refinement are well supported by the proof automation?
- How often can proof automation failures be overcome with user hints? What level of expertise is required to give effective hints?
- How helpful were the error feedback facilities during development?

9.2 Procedure

We now describe the procedure we used to carry out our case studies, where we aim to give a broad picture of how our framework can support dependently typed programming in practice.

9.2.1 Choice of Examples

We began the work in this chapter with a fixed list of the programs that needed to be developed. These were chosen so that we would be required to make use of a variety of data types and program properties so we could determine what was well supported by our framework. Most of the case studies are based on or inspired by example programs we have seen from current dependently typed programming languages. This is to show what support our framework provides for the kinds of programs current developers want to write and so that we can more easily make comparisons between the support our framework provides and the support provided in current environments. For example, the tail recursion case study is based on an example from ATS (see §9.3), the quicksort example is based on a Coq program example (see §9.3) and the binary adder is based on an Idris program (see §9.5).

9.2.2 Conducting a Case Study

Each case study involved implementing a dependently typed program and reporting on our experiences. The basic components of the instructions for each case study consisted of descriptions for the following:

Functionality: An informal description of the tasks the finished program should perform e.g. “implement an insertion sort function”.

Program properties: An informal description of the program properties that should be captured with the use of dependent types e.g. “verify that the list returned by the insertion sort function is always a permutation of the original list”.

When conducting the case studies, we intentionally avoided representations that the prototype has not been built to support. Specifically, we avoided the use of inductive predicates as these are not supported by our testing tool or our proof automation.

9.2.3 Reporting Case Studies

For each case study carried out, we give a factual account of the following:

- We describe the program written and the reason behind any relevant design choices. Standard function definitions used can be found in Appendix A.

- We describe how the proof automation performed when discharging proof obligations. We describe any proof attempt successes or failures worthy of note, but we do not exhaustively discuss each proof attempt. For brevity, unless otherwise stated, we describe the initial form of each proof obligation as being the form after exhaustive universal introduction is performed, pattern matching equations are substituted and subset type terms are destructured.
- When a proof obligation could not be automatically discharged, we describe our attempts to complete the proof with the use of the hinting feature and the various tactics our framework provides.
- We describe where the error feedback facilities were particularly helpful or unhelpful for developing certain kinds of programs. We did not attempt to exhaustively catalogue data regarding the errors we made during development.

When developing a program script, it is typical to first write one function, followed by another, followed by returning to modify the first function to correct an error. As a program script goes through many changes before it reaches its final form, it would be problematic to report on each and every proof obligation we encountered during case studies. Our pragmatic approach is that our description of the proof obligations are of those produced by the final script only. Moreover, whilst conducting a case study, when we were satisfied the specification of a function captured the property we had intended and we were convinced that the proof obligations generated by the function should be provable, we made no further modifications to that function definition.

9.2.4 System Configuration

The following describes how the system was configured when we conducted the case studies and why this configuration was chosen:

Initial lemmas: Each example program was developed from an empty proof script (i.e. cached lemmas were not shared between examples). Additionally, the lemma databases (see §7.2) at the start of each example were initially empty. This design choice was made to show how the prover copes without domain specific lemmas. Note that, for the purposes of describing the behaviour of our prover, we added a feature to make the prover report if a proof could only be found when cached lemmas were used.

Lemma discovery: We used a conservative configuration for the lemma discovery component (see §9.7.3), where only commutativity, associativity and involution properties were conjectured about each simply typed function definition used. We chose against checking for additional properties and only chose to check for these very common properties to avoid criticisms that the prover had been fine tuned to the examples.

Decision procedures: We note here that the prover does not make use of Coq's Presburger arithmetic procedure, which can easily be added as part of the `trivial` tactic. This choice was made to demonstrate how the prover would perform in environments without such a decision procedure, such as in Epigram.

Modifications: We only allowed for modifying the prototype during the case studies to patch easily fixable minor bugs that were preventing intended behaviour. Fortunately, such modifications were not required.

9.3 Case Study: Tail Recursive Functions

We now begin our description of the case studies that we conducted. We start by looking at a set of examples which involved writing efficient tail recursive functions. Each example involves making use of dependent types to verify that a tail recursive version of a function always computes the same result as a, simpler to define, naive definition. For each example function, we experiment with different possible representations, including the use of helper functions and higher-order fold functions. This case study was inspired from an example ATS program, where a tail recursive factorial function is verified [Xi, 2010].

9.3.1 List Sum

We start with the somewhat simpler task of implementing a function to sum a list of natural numbers. For this set of examples, our proof automation was able to discharge all of the 5 proof obligations that arose.

The standard naive definition of such a function is as follows:

```
Fixpoint sum (a:list nat) : nat :=
  match a with
  | [] ⇒ 0
  | h::t ⇒ h + sum t
  end.
```

9.3.1.1 Using a Helper Function

The first representation we use involves defining our tail recursive sum function using a helper function with an accumulator variable as follows:

```
Program Fixpoint sum_tail_aux (a:list nat) (acc:nat) :
  {r:nat | r = acc + sum a} :=
  match a with
  | [] ⇒ acc
  | h::t ⇒ sum_tail_aux t (acc + h)
  end.

Program Fixpoint sum_tail (a:list nat) :
  {r:nat | r = sum a} :=
  sum_tail_aux a 0.
```

Here we have used subset types to verify that sum_tail and sum always compute the same result.

The recursive call proof obligation of sum_tail_aux is as follows:

```
sum_tail_aux_p : sum_tail_aux_s = acc + h + sum t
______________________________________________________________
sum_tail_aux_s = acc + sum (h :: t)
```

Somewhat unexpectedly, the recursive_call pattern does not apply here as sum_tail_aux_p does not embed into the conclusion. As we commented in §6.5.1, this can happen when arguments in a recursive call are not all subterms of the corresponding arguments to the parent call. Here, the second argument of the recursive call is acc + h, which is not a subterm of acc.

As the output type of the recursive call here has the form {r | r = t}, where variable r does not occur in the term t, the propositional term produced by destructuring the recursive call will always be a non-recursive equation. The simplify tactic will always substitute with such equations. This goal thus simplifies to the following:

```
acc + h + sum t = acc + (h + sum t)
```

We find this pattern of simplification common to the solving the recursive call proof obligations in later examples of tail recursive programs where helper functions are used.

Returning to the goal above, as lemma discovery was already used to prove that + is associative, the prover trivially discharges this proof obligation. Finally, the sum_tail function generates no proof obligations as the return type for sum_tail_aux normalises to the expected type.

9.3.1.2 Using a Helper Function: Variant

To experiment with representation changes, we now take the program from the previous section and simply swap the order of the arguments to + in the output type and function body of sum_tail_aux. We would expect our framework to be able to support such a minor representation change seeing as the previous example was unproblematic.

This time, the recursive call proof obligation for sum_tail_aux simplifies to the following:

```
sum t + (h + acc) = h + sum t + acc
```

This variation of associativity is not yet known by the prover. The goal is discharged automatically by first generalising sum t and then performing a simple inductive proof.

9.3.1.3 Using Fold

Finally, we now attempt to define sum_tail using a fold function as follows:

```
Program Fixpoint sum_tail (a:list nat) :
  {r:nat | r = sum a} :=
  fold_left plus a 0.
```

This representation is pleasingly concise and similar to the style that is encouraged when writing regular functional programs. The above function generates the following proof obligation which, notably, contains a higher order function:

```
fold_left plus a 0 = sum a
```

The prover manages to discharge this proof obligation with the use of induction and lemma calculation (i.e. another inductive proof is required after fertilising then generalising the goal in the step case).

9.3.2 Factorial

We now perform a similar investigation into defining a tail recursive version of a factorial function, where we would expect to have to prove non-linear arithmetic properties. For this set of examples, our proof automation was able to automate 7 out of the 9 proof obligations that arose.

Capturing properties about a tail recursive factorial function has been seen previously in ATS examples [Xi, 2010], where manual proofs are required. Xi comments that it is "really tedious to establish [the] properties"[1] needed for this in ATS. We make use of the following function for the naive definition of factorial:

```
Fixpoint fact (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S p ⇒ S p ∗ fact p
  end
```

9.3.2.1 Using a Helper Function

Similar to before, we start by attempting to define a tail recursive version of fact with the use of a helper function and an accumulator variable as follows:

```
Program Fixpoint fact_tail_aux (n acc:nat) :
  {r:nat | r =  acc ∗ (fact n)} :=
  match n with
  | O ⇒ acc
  | S p ⇒ fact_tail_aux p (acc ∗ n)
  end.

Program Definition fact_tail (n:nat) : {r:nat | r = fact n} :=
  fact_tail_aux n 1.
```

The recursive call proof obligation of fact_tail_aux simplifies to the following:

```
acc ∗ S p ∗ fact p = acc ∗ (fact p + p ∗ fact p)
```

This goal is proven by first generalising the common subterm fact p and performing an inductive proof on acc.

[1] Personal communication.

9.3.2.2 Using a Helper Function: Variant

For this example, we copy the previous program script and swap the arguments to the $*$ operator in both the output type and in the function body of the fact_tail_aux function.

The recursive call proof obligation from fact_tail_aux now simplifies to the following, which turns out to be more challenging than before:

```
fact p * (acc + p * acc) = (fact p + p * fact p) * acc
```

The prover solves this goal by generalising the common subterm fact p and then performing an inductive proof on the fresh variable introduced by this step.

9.3.2.3 Using Fold

We now attempt a definition of a tail recursive factorial function that makes use of folding. To do this, we use the following function from Coq's standard library for generating a sequence of numbers:

```
Fixpoint seq (start len : nat) : list nat :=
  match len with
  | 0 ⇒ []
  | S p ⇒ start::seq (S start) p
  end.
```

For example, seq 2 4 is used to generate the list [2; 3; 4; 5]. A typical definition of factorial using fold is as follows, where mult is the function that $*$ is annotation for:

```
Program Definition fact_tail (n:nat) : {r:nat | r = fact n} :=
  fold_left mult (seq 1 n) 1.
```

This function generates the following simplified proof obligation:

```
fold_left mult (seq 1 n) 1 = fact n
```

Unfortunately, the prover fails to prove this goal. Positive progress towards a proof is made however. Induction on n yields a trivial base case and in the step case rippling can fully ripple the RHS. Lemma calculation then conjectures the following theorem:

```
∀ n, fold_left mult (seq 2 n) 1 =
  fold_left mult (seq 1 n) 1 + n * fold_left mult (seq 1 n) 1
```

The prover is unable to prove this goal and we could see no obvious hints that could help.

9.3.2.4 Another Attempt Using Fold

We next tried reimplementing the body of fact_aux using fold, instead of fact_tail , to see how the prover copes. We predicted that this proof should be easier to automate. We defined fact_aux as follows:

```
Program Definition fact_aux (n acc:nat) :
  {r:nat | r = acc * fact n} :=
  fold_left mult (seq 1 n) acc.
```

The proof obligation generated from this function is as follows:

```
fold_left mult (seq 1 n) acc = acc * fact n
```

Unfortunately, this again turned out to be too difficult for the prover to discharge.

9.3.3 Inorder Tree Traversal

As a final example in this case study, we consider writing an optimised version of the following inorder traversal function for binary trees:

```
Fixpoint inorder (a:btree A) : list A :=
  match a with
  | empty ⇒ []
  | node v l r ⇒ (inorder l) ++ [v] ++ (inorder r)
  end.
```

For this set of examples, our automation was able to discharge 4 out of the 5 proof obligations that arose.

9.3.3.1 Using a Helper Function

As there are two recursive calls, there is no simple way to write this function using tail recursion. We can however settle for a definition where the left subtree is traversed with tail recursion and use an accumulator to replace the expensive ++ operator with ::. This definition, defined with a helper function, is as follows:

```
Program Fixpoint inorder_aux (a:btree A) (acc:list A) :
  {r:list A | r = inorder a ++ acc} :=
  match a with
  | empty ⇒ acc
  | node v l r ⇒ inorder_aux l (v::(inorder_aux r acc))
  end.

Program Fixpoint inorder_tail (a:btree A) :
  {r:list A | r = inorder a} :=
  inorder_aux a [].
```

The base case proof obligation of inorder_aux is proven by reflexivity. The recursive call proof obligation simplifies to the following:

```
inorder l ++ v :: inorder r ++ acc =
  (inorder l ++ v :: inorder r) ++ acc
```

The proof proceeds by generalising the common subterms inorder l and inorder r to produce the following:

```
c1 ++ v :: (c2 ++ acc) = (c1 ++ v :: c2) ++ acc
```

This goal is then proven with a simple inductive proof. Finally, the proof obligation generated by inorder_tail is trivially automated.

9.3.3.2 Using Fold

We move onto defining a version of inorder_tail that uses fold. We implement such a function as follows, where fold here performs an inorder traversal of a tree:

```
Program Fixpoint inorder_tail (a:btree A) :
  {r:list A | r = inorder a} :=
  fold (fun acc v ⇒ v::acc) a [].
```

Notice that a λ term features in this program. The proof obligation generated by this function is as follows:

```
fold (fun acc v ⇒ v :: acc) a [] = inorder a
```

Unfortunately, the prover is unable to automate this proof. After induction on variable a, lemma calculation results in the following goal, which the prover is unable to solve:

```
fold_right (fun acc v ⇒ v :: acc) b l =
fold_right (fun acc v ⇒ v :: acc) b [] ++ l
```

9.3.3.3 Using Fold: Variant

This time we attempt to reimplement inorder_aux using a fold as follows:

```
Program Fixpoint inorder_aux (a:btree A) (acc:list A):
  {r:list A | r = inorder a ++ acc} :=
  fold (fun acc v ⇒ v::acc) a acc.
```

The proof obligation generated by inorder_aux is as follows, where, for brevity, we have replaced the lambda term with f:

```
fold f a acc = inorder a ++ acc
```

The prover succeeds at finding a proof. The proof begins by induction on the tree a, where the step case goal produces multiple givens. The following shows the step case, where we display one annotated conclusion for each hypothesis:

```
H1 : ∀ acc, fold f r acc = inorder r ++ acc
H2 : ∀ acc, fold f l acc = inorder l ++ acc
```

$$\text{fold } f\ (\colorbox{lightgray}{$\text{node } v\ l\ \colorbox{white}{r}$}^{\uparrow})\lfloor acc \rfloor = \text{inorder } (\colorbox{lightgray}{$\text{node } v\ l\ \colorbox{white}{r}$}^{\uparrow})\ {+}{+} \lfloor acc \rfloor$$
$$\text{fold } f\ (\colorbox{lightgray}{$\text{node } v\ \colorbox{white}{l}\ r$}^{\uparrow})\lfloor acc \rfloor = \text{inorder } (\colorbox{lightgray}{$\text{node } v\ \colorbox{white}{l}\ r$}^{\uparrow})\ {+}{+} \lfloor acc \rfloor$$

After computing with the fold function, the LHS is rippled out to the following:

$$\colorbox{lightgray}{$\text{fold } f\ l\ (v :: \colorbox{white}{$\text{fold } f\ r \lfloor acc \rfloor$}$}^{\uparrow} = \text{inorder } (\colorbox{lightgray}{$\text{node } v\ l\ \colorbox{white}{r}$}^{\uparrow})\ {+}{+} \lfloor acc \rfloor$$
$$\text{fold } f\ l\ \lfloor v :: \text{fold } f\ r\ acc \rfloor = \text{inorder } (\colorbox{lightgray}{$\text{node } v\ \colorbox{white}{l}\ r$}^{\uparrow})\ {+}{+} \lfloor acc \rfloor$$

Weak fertilisation proceeds by rewriting the LHS with H1 and then H2 in sequence. After this, lemma calculation is then used to finish the proof. Compared to the failed proof attempt in the previous section, weak fertilisation is possible this time because of the presence of the acc sink in the conclusion.

9.3.4 Error Feedback

We found the error feedback particularly useful for developing this style of examples. In particular, as these examples used strong specifications, we found the error messages produced easy to follow (we mentioned in §8.1.3 that weak specifications can make the error messages harder to understand). The following gives some examples of the ways the error feedback helped:

- When giving the output type of the helper functions, it is relatively easy to treat the accumulator variable incorrectly. For example, we initially wrote r = acc + (fact n) instead of r = acc $*$ (fact n) in the output type for fact_aux. Likewise, we accidentally wrote r = acc ++ inorder a instead of r = inorder a ++ acc in the output type for inorder_aux. In each case, we were alerted to the error and examining the counterexample trace made the cause of the problem obvious.
- When defining fact, we accidentally made the base case return 0 instead of 1. This became obvious when examining a trace of a counterexample found when defining the first fact_aux function.

9.4 Case Study: Insertion Sort, Tree Sort and Quicksort

In this section, we make use of subset types for capturing properties of programs that implement insertion sort, tree sort and quicksort. We explore what support can be given for capturing length and permutation properties.

Note that, for ease of presentation, the implementations given are specialised for collections of natural numbers. We make use of the following comparison function in each sorting program:

```
le_gt_dec : ∀ n m : nat, {n ≤ m} + {n > m}
```

9.4.1 Insertion Sort

The first sorting algorithm that we verify is insertion sort. Our automation managed to discharge all 6 of the proof obligations that arose for this example.

The first property that we wish to capture is that the insertion sort function we implement returns a list of the expected length. Such a program can be implemented in a straightforward way as follows, where we have used subset types to capture the length of the sorted output list:

```
Program Fixpoint insert (x:nat) (a:list nat) :
  {r:list nat | length r = S (length a)} :=
  match a with
  | nil ⇒ [x]
  | h::t ⇒ if le_gt_dec x h then x::a else h::(insert x t)
  end.

Program Fixpoint insertion_sort (a:list nat) :
  {r:list nat | length r = length a} :=
  match a with
  | nil ⇒ nil
  | h::t ⇒ insert h (insertion_sort t)
  end.
```

The insert function adds an item into a sorted list such that the is also sorted. The insertion_sort function recursively adds each item from an unsorted list a into an initially empty list using insert such that the resultant list from insertion_sort is a sorted permutation of a.

9.4.1.1 Length Property

We now consider the proof obligations produced by the program above, ignoring the more trivial ones. The recursive call proof obligation produced by insert is as follows:

```
insert_p : length insert_s = S (length t)
________________________________________________
length ( h :: insert_s ↑) = S (length ( h :: t ↑))
```

The proof for this goal follows the recursive_call proof pattern and is discharged automatically.

Notice that the insertion_sort function contains a call to insert, where insert returns a subset type. The recursive call proof obligation produced by insertion_sort, without destructuring the subset type terms, is as follows:

```
length (proj1_sig (insert h (proj1_sig (insertion_sort t)))) =
  length (h :: t)
```

To prove this goal, the prover follows the recursive_call pattern. This first involves destructuring the call to only the recursive call term, producing the following goal:

```
e : length x = length t
________________________________________________
length ( proj1_sig (insert h x ) ↑) = length ( h :: t ↑
```

The prover then fully ripples out and weak fertilises the RHS to produce the following goal:

```
length (proj1_sig (insert h x)) = S (length x)
```

The prover finishes the proof by destructuring the result of insert and directly applying the propositional term produced to prove the conclusion.

Note that, if the prover had blindly destructured both of the subset type terms, the following goal would have been produced:

```
insertion_sort_p : length insertion_sort_s = length t
insert_p : length insert_s = S (length insertion_sort_s)
________________________________________________________________
length insert_s = length (h :: t)
```

Notice that there are no embeddings in this goal so a more ad hoc and less guided approach would have been needed to solve this goal, compared to the use of rippling.

9.4.1.2 Length Property: Variation

Next, we consider what happens when the insert function is changed to being simply typed, where it no longer specifies the length of the list it returns. By making this change, we can see how the framework copes when the programmer decides to make use of functions with less informative types. From experience, we know that this can make the proofs involved more challenging.

This time, the recursive call proof obligation produced by insertion_sort only contains one subset type term (i.e. the call to insertion_sort). The recursive_call pattern is followed where, after rippling out and weak fertilising, the following goal is produced:

```
length (insert h insertion_sort_s) =
  S (length insertion_sort_s)
```

The prover discharges this goal by induction over insertion_sort_s . Notice that the goal above encodes the information that we chose to specify by hand in the output type of insert in the previous section. We see here that the convenience of fewer annotations can result in more challenging proofs and, in this case, the prover can support both representations.

9.4.1.3 Permutation Property

We now consider capturing the property that insertion_sort returns a permutation of its input. To represent a permutation, we make use of the following function which returns the number of terms in a list that have the same value as x:

```
Fixpoint list_count (a:list nat) (x:nat) : nat :=
  match a with
  | nil ⇒ O
  | h::t ⇒ if nat_eq_dec h x then
              S (list_count t x)
            else
              list_count t x
  end.
```

We use list_perm x y as shorthand for ∀ n, list_count x n = list_count y n to represent that list x is a permutation of list y. We chose this representation as it was simple to define and generally useful for capturing other properties about lists.

To create our program, we simply copy the one from §9.4.1.1 and replace the length propositions with propositions concerning permutation properties as follows:

```
Program Fixpoint insert (x:nat) (a:list nat) :
  {r:list nat | list_perm r (x::a)} :=
  (* as before *)

Program Fixpoint insertion_sort (a:list nat) :
  {r:list nat | list_perm r a} :=
  (* as before *)
```

The recursive call proof obligation generated by the insert function, which can be automatically discharged, has the following form:

$$\frac{\text{insert\_p} : \forall\, \text{n} : \text{nat},\ \text{list\_count}\ \ \text{insert\_s}\ \ \text{n} = \text{list\_count}\ \ (\text{x} :: \text{t})\ \text{n}}{\text{list\_count}\ \ (\boxed{\text{h} :: \boxed{\text{insert\_s}}}^{\uparrow})\lfloor \text{n} \rfloor = \text{list\_count}\ \ (\text{x} :: \boxed{\text{h} :: \boxed{\text{t}}}^{\uparrow})\lfloor \text{n} \rfloor}$$

Similarly to when the length property of this function was captured, this goal contains embeddings and the proof again involves following the recursive call pattern. In this case, the goal features sinks, because list_perm includes a universal quantifier. Additionally, the conditional statement used to define list_count means that a case split must be performed before weak fertilisation can occur.

The recursive call proof obligation produced by insertion_sort follows a similar course to before with the use of the recursive call pattern. Again, the difference here is that a case split is required before weak fertilisation can occur.

9.4.1.4 Permutation Property: Variation

For this example, we modified our previous program such that insert was now a simply typed function instead of one that returns a subset type to produce the following program (notice that insertion_sort still returns a subset type):

```
Fixpoint insert (x:nat) (a:list nat) : list nat :=
  (* as before *)

Program Fixpoint insertion_sort (a:list nat) :
  {r:list nat | list_perm r a} :=
  (* as before *)
```

Again, the prover is able to automate all the proofs required. As with the previous program, the proof for the recursive call proof obligation generated by insertion_sort follows the recursive_call pattern. Lemma calculation is required to finish this proof, where the following two lemmas are proven and cached in the process:

```
1) list_count (insert n insertion_sort_s) n =
     S (list_count insertion_sort_s n)

2) h ≠ n →
     list_count (insert h insertion_sort_s) n =
       list_count insertion_sort_s n
```

The second lemma is automatically identified as a right-to-left simplification rule (see §7.11). It was not immediately obvious to us that this was a useful simplification rule, but we agreed with the classification on inspection.

9.4.2 Tree Sort

We now look at implementing a program to sort lists using *tree sort*. To sort a list with this algorithm, items from an unsorted list are first inserted one by one into a binary tree. A list created by performing an inorder traversal of this tree will result in a sorted permutation of the original list. For this set of examples, our automation discharged 10 out of the 12 proof obligations that arose.

To implement tree sort, we use the simple type btree to represent binary trees (see §A.3). The following gives a straightforward implementation of tree sort, where subset types are used to capture the length property of the resulting sorted list:

```
Program Fixpoint insert (x:nat) (a:btree nat) :
  {r:btree nat | num_nodes r = num_nodes (node x a empty)} :=
  match a with
  | empty ⇒ node x empty empty
  | node y l r ⇒
    if le_gt_dec x y then
      node y (insert x l) r
    else
      node y l (insert x r)
  end.

Program Fixpoint sorted_tree_of_list (a:list nat) :
  {r:btree nat | num_nodes r = length a} :=
  match a with
  | [] ⇒ empty
  | h::t ⇒ insert h (sorted_tree_of_list t)
  end.

Program Fixpoint tree_sort (a:list nat) :
  {r:list nat | length r = length a}:=
  inorder (sorted_tree_of_list a).
```

The insert function insert an item into a sorted binary tree. Note that we do not consider balanced trees in this implementation. The sorted_tree_of_list function converts an unsorted list into a sorted binary tree. The tree_sort function first converts the input unsorted list into a sorted binary tree and then returns the inorder traversal of this tree to give the final sorted list. The simply typed functions inorder and num_nodes are used to return the inorder traversal of a tree and the number of nodes in a tree respectively.

9.4.2.1 Length Property

We begin by considering the proof obligations produced by the program above, where we have captured the length property of the final sorted list. Compared to the insertion sort algorithm in the last set of examples, the proofs involved this time naturally require reasoning about trees, as well as lists.

For the insert function, there are two recursive call proof obligations to discharge. In each case, the proof is automated by an inductive proof of a simple linear arithmetic property after the use of the recursive_call pattern.

For the sorted_tree_of_list function, the proof of the recursive call proof obligation involves using the propositional terms returned by both the recursive calls and the call to insert. In this sense, this proof has similarities to the proof for the recursive call proof obligation for the insertion_sort function from §9.4.1.1. For this proof, the recursive call terms are destructured and rippling is used to weak fertilise the goal with the propositional terms generated. When the result from insert is then destructured, the goal has the following form:

```
insert_p : num_nodes insert_s = num_nodes (node h x empty)
-----------------------------------------------------------
num_nodes insert_s = S (num_nodes x)
```

The proof is completed by first using the cross_fertilise tactic to rewrite the conclusion with insert_p from left to right. Simplification, generalisation and induction are then used to finish the proof.

The tree_sort function produces one proof obligation. Here, the cross_fertilise tactic is used to fertilise the conclusion with the propositional term returned by the sorted_tree_of_list function, resulting in the following goal:

```
length (inorder sorted_tree_of_list_s) =
  num_nodes sorted_tree_of_list_s
```

As the inorder function is simply typed, induction is naturally needed to prove this goal. This goal is proven by induction over the variable sorted_tree_of_list_s .

9.4.2.2 Permutation Property

Reusing the previous program, we now capture the property that the output list from tree_sort is a permutation of the input list to the function. To do this, we reuse the list_count function and list_perm notation from §9.4.1.3. We introduce the notation btree_perm x y to denote that tree x is a permutation of tree y. This notation is shorthand for ∀ n, btree_count x n = btree_count y n, where btree_count x n returns the number of terms with the same name as n in tree x. The btree_count function is defined as follows:

```
Fixpoint btree_count (a : btree nat) (x : nat) : nat :=
  match a with
  | empty ⇒ O
  | node v l r ⇒
    let countlr := btree_count l x + btree_count r x in
    if nat_eq_dec v x then (S countlr) else countlr
  end.
```

This function is more complex than count_list in that the member of both subtrees must be considered. We modify the output types of the tree sort implementation as follows:

```
Program Fixpoint insert (x:nat) (a:btree nat) :
  {r:btree nat | btree_perm r (node x a empty)} :=
  (* as before *)

Program Fixpoint sorted_tree_of_list (a:list nat) :
  {r:btree nat | ∀ n, btree_count r n = list_count a n} :=
  (* as before *)

Program Fixpoint tree_sort (a:list nat) :
  {r:list nat | list_perm r a}:=
  (* as before *)
```

We now consider the proof obligations produced by this program. The proofs required for the sorted_tree_of_list and insert functions have a similar shape to the ones required in the previous section. This time, the proofs are more complex in that case splits are performed during rippling.

Unfortunately, the prover fails to automate the proof obligation generated by the tree_sort function. After simplification, the proof obligation has the following form:

```
list_count (inorder sorted_tree_of_list_s) n =
  btree_count sorted_tree_of_list_s n
```

After induction on variable sorted_tree_of_list_s , a case split is performed in the step case proof and lemma calculation results in the following goal, which the prover is unable to automate:

```
list_count (inorder l ++ [n] ++ inorder r) n =
 S (list_count (inorder l) n + list_count (inorder r) n)
```

A productive generalisation here is to generalise the inorder l and inorder r common subterms as follows:

```
∀ x y n, list_count (x ++ [n] ++ y) n
 S (list_count x n + list_count y n)).
```

The generalise tactic makes this step but, unfortunately, follows on by overgeneralising the goal by generalising apart the occurrences of the variable n. The overgeneralisation is detected by our testing tool, but this event causes all the generalisation steps to be undone (see §7.5.6). As the alternative proof path is to attempt induction on the ungeneralised conjecture, the prover eventually fails.

To work around this, after identifying from the proof search trace that this goal was not being generalised correctly, we provided the above correct generalisation as a hint. The prover successfully proved this lemma and then managed to discharge the failing proof obligation by using this new lemma to trivially prove the problematic goal.

9.4.2.3 Permutation Property: Variation

The output type of the previous definition of sorted_tree_of_list checks that the output tree contains the same elements as the input list by making use of tree_count and list_count . To experiment with different representation choices, we now consider the following alternative representation for the output type:

```
Program Fixpoint sorted_tree_of_list (a:list nat) :
 {r:btree nat | list_perm (inorder r) a} :=
 (* as before *)
```

This time for the output type, the output tree is converted to a list using inorder and we then check that this list is a permutation of the input list.

Unfortunately, the prover is unable to automate the recursive call proof obligation generated by this function. Briefly, the proof attempt follows the recursive_call pattern, where rippling produces two subgoals. The prover is then unable to automate either of these subgoals. We consider only the first subgoal here, which is as follows:

```
insert_p : ∀ n : nat, btree_count insert_s n =
                      btree_count (node n x empty) n
----------------------------------------------------------
list_count (inorder insert_s) n = S (list_count (inorder x) n)
```

Rippling does not apply and the simplify tactic cannot do anything productive. Any successful inductive proof would likely require piecewise fertilisation [Armando et al., 1999], which the current system does not support.

To work around this, we considered how the goal above could be modified to allow fertilisation with insert_p. We reasoned that the following lemma, when used from left to right to rewrite the conclusion, would allow this:

```
L : ∀ x n, list_count (inorder x) n = btree_count x n
```

We asked the system to prove L but the automation failed. Notice that L has the same form as the proof obligation that could not be fully automated in the previous section because of an overgeneralisation occurring in the proof attempt. We thus provided the same generalisation hint from the previous section (recall that we are not sharing cached lemmas between example programs) and this allowed L to be proven.

After adding L as a left to right simplification rule, the problematic goal above was then successfully automated. The automation succeeded this time because the simplify tactic was able to alter the goal of the conclusion so that fertilisation could occur and induction was then used to finish the proof. We note that this solution is not ideal because converting uses of list_count to tree_count is not always going to be desirable in every proof.

Finally, unlike in §9.4.2.2, the proof obligation produced by tree_sort is trivially automated with the use of simplification.

9.4.3 Quicksort

As the final set of examples in this case study, we now consider sorting with the well known and efficient quicksort algorithm. Our automation was only moderately successful for this example by being able to discharge 5 out of the 10 of the proof obligations that arose.

Quicksort is traditionally not written in a structurally recursive manner and can be problematic to define in languages where all function definitions are required to terminate. Sozeau's Program tactic includes a feature that allows a non-structurally recursive function to be defined if a decreasing measure is provided (see §3.3.4). This feature can be used to give a natural definition of quicksort in Coq. For this set of examples, we have adapted an implementation of quicksort written by Sozeau[2] that uses this feature.

In his proof script, Sozeau captures the full specification of quicksort using subset types. Unfortunately, the approach used to do this is not well supported by our framework. Specifically, the propositional parts of the subset types are defined with the use of inductive predicates. We noted earlier in §7.14.3, that working with such a representation is not currently supported by our prototype. We therefore chose to adapt this quicksort implementation, maintaining the same program structure (and making some minor cosmetic differences) but changing the way the propositional statements were expressed. Our adaptation, where the length property of the resulting list has been captured, is as follows:

```
Program Fixpoint split (pivot:nat) (a:list nat) :
    {(lower, higher):list nat * list nat |
      length lower + length higher = length a} :=
  match a with
  | nil ⇒ ([], [])
  | h::t ⇒
    match split pivot t with
    | (lower, higher) ⇒
      if le_gt_dec h pivot then
        (h::lower, higher)
      else
        (lower, h::higher)
    end
  end.
```

[2]Available at http://mattam.org/repos/coq/misc/sort/quicksort.v

```
Program Fixpoint quicksort (a:list nat) {measure (length a)} :
    {r:list nat | length r = length a} :=
  match a with
  | [] ⇒ []
  | h::t ⇒
    match split h t with
    | (lower, higher) ⇒ quicksort lower ++ [h] ++ quicksort
      higher
    end
  end.
```

The split function splits a list into two sublists based on a pivot item, where members of the first list are all lower than the pivot and members of the second list are all higher than the pivot. The quicksort function, using the head value as the pivot, uses split to partition a list in two and then applies the quicksort function again on these two sublists. The decreasing measure specified (using the measure keyword) is that the length of the output from each recursive call to quicksort is always less than the length of the input for this call.

9.4.3.1 Length Property

We begin by considering the proof obligations generated by the above program. Firstly, the proof obligations from the split function are successfully automated, where proofs of simple arithmetic properties are required.

The recursive call proof obligation generated by the quicksort function is interesting in that it contains two recursive call terms. The usual strategy of following the recursive_call pattern fails here as no embeddings are found between the conclusion and the proofs terms generated from these recursive calls. This is perhaps unsurprising when we remember that quicksort has not been defined by structural recursion. The proof thus proceeds by destructing all subset type terms to produce the following goal:

```
quicksort1_p : length quicksort1_s = length higher
quicksort2_p : length quicksort2_s = length lower
split_p : length lower + length higher = length t
─────────────────────────────────────────────────────────
length (quicksort2_s ++ [h] ++ quicksort1_s) = length (h :: t)
```

As there are no embeddable assumptions to ripple with here, the simplify tactic proceeds by simplifying the RHS of the conclusion and then rewriting the RHS of the conclusion with the split_p , quicksort1_p and quicksort2_p equations from right-to-left in sequence to give the following:

```
length (quicksort2_s ++ h :: quicksort1_s) =
  S (length quicksort2_s + length quicksort1_s)
```

This goal is then discharged automatically with a simple inductive proof over variable quicksort2_s.

We now consider the proof required to show that quicksort always terminates. We must prove the supplied measure is decreasing for both of the recursive calls to quicksort. The first call to quicksort requires a proof of the following:

```
length lower < S (length lower + length higher)
```

The prover is unable to solve this goal as it attempts to perform a proof by induction but is unable to reason about the inductive predicate $<$. The second measure proof obligation also has a similar shape to the above. Note that we can work around these proof automation failures by manually calling Coq's Presburger arithmetic procedure.

9.4.3.2 Permutation Property

We now capture the property that the result of the quicksort function is a permutation of its input. We do this by changing the output types of the previous program to the following:

```
Program Fixpoint split (pivot:nat) (a:list nat) :
    {(lower, higher):list nat * list nat |
      list_perm (lower ++ higher) a} :=
  (* as before *)

Program Fixpoint quicksort (a:list nat) {measure (length a)} :
    {r:list nat | list_perm r a} :=
  (* as before *)
```

The proof obligations generated by the split function are both successfully automated. Of note, when discharging the proof obligation generated by the second **if** clause in this function, the following lemma is cached:

```
count_simp : ∀ h x y n,
h ≠ n → list_count (x ++ h :: y) n = list_count (x ++ y) n
```

This lemma was then automatically added to the simplification lemma database as a left to right simplification rule as the RHS of the equation embeds into the LHS (see §7.11). Note that, to produce this reusable lemma, delayed generalisation (see §7.10) was used to remove irrelevant assumptions, such as pivot $<$ h, from the original proof found for this lemma. This simplification rule becomes relevant for discharging the next proof obligation.

The recursive call proof obligation for the quicksort function has a similar shape to what was seen in the previous section. This time, however, this cannot be automated without help. After destructuring the subset type terms, the proof obligation here has the following form:

```
quicksort1_p : ∀ n, list_count quicksort1_s n = list_count higher n
quicksort2_p : ∀ n, list_count quicksort2_s n =list_count lower n
split_p : ∀ n, list_count (lower ++ higher) n = list_count t n
──────────────────────────────────────────────────────────────────
list_count (quicksort2_s ++ [h] ++ quicksort1_s) n = list_count (h :: t) n
```

Again, there are no embeddings to ripple with. The simplify tactic proceeds by simplifying the conclusion and then performing a case split on the **if** construct produced on the RHS to give two subgoals. In both subgoals, the split_p equation is used to rewrite the RHS of the conclusion from right to left. Unlike last time, quicksort1_p and quicksort2_p cannot yet be used to rewrite the conclusion. Somewhat surprisingly however, these two terms now embed into the conclusion. At this stage in the proof attempt, the second subgoal has the following form:

```
quicksort1_p : ∀ n, list_count quicksort1_s n = list_count higher n
quicksort2_p : ∀ n, list_count quicksort2_s n = list_count lower n
c : h ≠ n
──────────────────────────────────────────────────────────────────
```

$$\text{list\_count}\ (\boxed{\text{quicksort2\_s} +\!+\ \text{h} :: \boxed{\text{quicksort1\_s}}}^{\uparrow})\lfloor \text{n} \rfloor = \text{list\_count}\ (\boxed{\text{lower} +\!+ \boxed{\text{higher}}}^{\uparrow})\lfloor \text{n} \rfloor$$

$$\text{list\_count}\ (\boxed{\boxed{\text{quicksort2\_s}} +\!+\ \text{h} :: \text{quicksort1\_s}}^{\uparrow})\lfloor \text{n} \rfloor = \text{list\_count}\ (\boxed{\boxed{\text{lower}} +\!+ \text{higher}}^{\uparrow})\lfloor \text{n} \rfloor$$

Notice that the LHS of the conclusion matches the RHS of the cached simplification rule count_simp found at the start of this section, where assumption c is a proof of the necessary side-condition. The simplify tactic continues its work by using count_simp to eliminate the h :: term from the conclusion. Unfortunately, instead of guiding the proof with rippling, the prover then tries to perform an inductive proof and fails. This happens here because the prover never expects to see any terms that embed into the conclusion between its simplification and induction steps (see §7.1).

Noticing from the proof trace that the propositional terms from the recursive call were not being used, we attempted a manual proof. We invoked the simplify tactic and, upon noticing the embeddings in each subgoal, called the rippling tactic manually. However, rippling was immediately blocked. We found it easy to see from the rippling annotations that the following rule would unblock this rippling proof:

```
∀ x y n, list_count (x ++ y) n = (list_count x n) + (list_count
   y n)
```

After being supplied with the above statement in the form of a hint, the prover was able to use this to finish the proof via rippling automatically.

A Challenging Termination Proof

When capturing only the length property of the quicksort program in §9.4.3.1, the termination proof obligation could not be automated by the prover but it was possible to work around this by simply invoking Coq's Presburger arithmetic procedure manually. For this program, the proof required is more challenging. For the first recursive call to quicksort, we are required to prove the following goal:

```
split_p : list_perm (lower ++ higher) t
--------------------------------------------------------
length lower < S (length t)
```

The prover is unable to make any useful progress on this goal so we have to resort to a manual proof. The approach we used was to first prove the following lemmas:

```
perm_length : ∀ x y, list_perm x y → length x = length y
length_app : ∀ x y, length (x ++ y) = length x + length y
```

We can then manually use perm_length with assumption split_p to produce a proof of length (lower ++ higher) = length t. After rewriting this new assumption using the lemma length_app from right to left , the goal can be discharged by invoking Coq's Presburger arithmetic procedure. The prover can help in this manual proof in that it can automatically prove length_app for us when asked to do so. However, the prover cannot automate the proof for perm_length. The second recursive call to quicksort requires a similar termination proof.

9.4.4 Error Feedback

We again found the error feedback helpful for quickly identifying errors when developing this set of examples but did experience some problems. As the testing tool cannot test goals that have assumptions that contain universal quantifiers, error feedback could not be given for most of the examples where we captured permutation properties. Specifically, this was the case when the tree_perm and list_perm notations were being used. For instance, when the prover failed to prove the goal from §9.4.3.2, we were less sure the goal was provable when testing was not available. However, as the permutation property examples were created by modifying the output types of previous examples (where the length property was captured first), the lack of error feedback for the former examples was less of a problem in practice.

When capturing the length properties, the feedback was generally useful in alerting us to problems and the error messages produced offered some help in fixing the error messages. As we noted in §8.1.3, some thought is needed when interpreting the error messages when weak specifications are being used, which is the case for the length property examples.

9.5 Case Study: Binary Adder

In this case study, we port a program written in Idris to Coq to see what support can be provided. This Idris program[3] makes use of inductive families to verify that a binary adder performs as expected [Brady, 2008]. We chose this example as the program makes an interesting use of inductive families, non-linear arithmetic properties are involved and the comments in Brady's program script implies that the proofs required to define the program were tedious to write. Note that, due to some bugs we encountered in the Program tactic, some of the Russell functions in this section had to be written as regular Coq functions. Our automation was able to discharge all but one of the 20 proof obligations that arose in this case study.

9.5.1 Inductive Families Representation

We start by introducing all the data types that will be needed in the main program. The following type represents a binary bit indexed by its natural number representation:

[3]Available at `http://www.cs.st-andrews.ac.uk/~eb/drafts/binary.idr`

```
Inductive Bit : nat → Set :=
  | bit0 : Bit 0
  | bit1 : Bit 1.
```

For example, we know from the type index that bit0 represents that nat value 0. The following type, indexed in the same way, represents a pair of bits, where the leftmost bit is taken as the most significant bit:

```
Inductive BitPair : nat → Set :=
  | bitPair : ∀ c v, Bit c → Bit v → BitPair (v + 2 ∗ c).
```

For example, bitPair bit1 bit0 has type BitPair (0 + 2 ∗ 1). The type can be interpreted as “a bit pair whose decimal value is 2“. The following type represents a binary number, composed of Bit terms, where the type is indexed by its natural number representation as well as its length:

```
Inductive Number : nat → nat → Set :=
  | none : Number 0 0
  | bit : ∀ b n val, Bit b → Number n val →
           Number (S n) ((2 ^ n) ∗ b + val).
```

For example, the type Number 8 32 represents “a binary number composed of 8 bits that has the decimal value 32”. Similarly indexed, the following type represents a binary number coupled with a carry bit:

```
Inductive NumCarry : nat → nat → Set :=
  | numCarry : ∀ c n val, Bit c → Number n val →
                 NumCarry n ((2 ^ n) ∗ c + val).
```

We now define several utility functions before defining the binary adder function. The first function adds a pair of bits x to the leftmost position of a binary number num:

```
Program Fixpoint msPair
  (b n val:nat) (x:BitPair b) (num:Number n val) :
  NumCarry (S n) ((2 ^ n) ∗ b + val) :=
  match x with
  | (bitPair _ _ c v)  ⇒ numCarry c (bit v num)
  end.
```

We must discharge a proof obligation to show that the binary result has the expected natural number representation. The conclusion of the proof obligation generated is as follows after simplifying by performing computations:

```
(2^n + (2^n + 0)) * s + (2^n * t + val) = 2^n * (t + (s + (s + 0))) + val
```

The prover is able to discharge this proof obligation automatically. In the proof, the simplify tactic simplifies the goal using the rule ∀ x, x + 0 = x. This rule was found during lemma discovery and automatically added as a simplification rule. The 2^n term is then identified as a common subterm and generalised to produce the following:

```
(c + c) * s + (c * t + val) = c * (t + (s + s)) + val
```

The prover then automates this proof via induction on c, with lemma calculation being needed several times. This proof is challenging in that arithmetic lemmas found during lemma discovery are required to find a proof.

The initial steps of Brady's hand written proof for the above similarly involves making the same simplifications and generalising the 2^n term. However, instead of induction, Brady makes use of commutativity, associativity and distributive theorems about + and ∗ to rewrite the goal to finish the proof.

The next function we need to define sums together three bits labelled x, y and z, returning a pair of bits:

```
Program Definition addBit
  (r l c:nat) (x:Bit c) (y:Bit l) (z:Bit r) :
  BitPair (c + (l + r)) :=
  match x, y, z with
  | bit0 , bit0 , bit0 ⇒ bitPair bit0 bit0
  | bit0 , bit0 , bit1 ⇒ bitPair bit0 bit1
  | bit0 , bit1 , bit0 ⇒ bitPair bit0 bit1
  | bit0 , bit1 , bit1 ⇒ bitPair bit1 bit0
  | bit1 , bit0 , bit0 ⇒ bitPair bit0 bit1
  | bit1 , bit0 , bit1 ⇒ bitPair bit1 bit0
  | bit1 , bit1 , bit0 ⇒ bitPair bit1 bit0
  | bit1 , bit1 , bit1 ⇒ bitPair bit1 bit1
  end.
```

This function is simply implemented as a lookup table, where no proof obligations are generated. The next function adds the two bits x and y to a binary number with a carry bit nc:

```
Program Fixpoint addNumberAux
  (l r n val:nat) (x:Bit l) (y:Bit r) (nc:NumCarry n val) :
```

```
  NumCarry (S n) ((2 ^ n) * (l + r) + val) :=
  match nc with
  | numCarry _ _ _ c num ⇒ msPair (addBit x y c) num
  end.
```

This function generates the following proof obligation:

```
2^n * (l + (r + c)) + val = 2^n * (l + r) + (2^n * c + val)
```

The proof found by the prover is similar to the proof required for the msPair function, where the 2^n term is first generalised and induction is used to finish the proof. Brady's proof again involves the same generalisation and makes use of rewriting instead of induction.

We now consider the function that sums two binary numbers. This function is defined to only accept two binary numbers that have the same length, where summing is performed recursively by adding together the leftmost bits of the two numbers:

```
Program Fixpoint addNumber
  (n l r c:nat) (x:Number n l) (y:Number n r) (b:Bit c) :
  NumCarry n (c + (l + r)) :=
  match x, y with
  | bit _ _ _ _ _, none ⇒ !
  | none, bit _ _ _ _ _ ⇒ !
  | none, none ⇒ numCarry b none
  | bit _ _ _ b1 num1, bit _ _ _ b2 num2 ⇒
      addNumberAux b1 b2 (addNumber num1 num2 b)
  end.
```

The first two match clauses are marked as impossible cases as the numbers being added must be the same length. All the proof obligations generated by this function were discharged automatically by our top-level tactic. The proof obligations generated by the first three match clauses are trivially automated. The final match clause produces the following proof obligation, where some simplification has already been performed:

```
2^n * (x + y) + (c + (l + r)) = c + (2^n * x + l + (2^n * y + r))
```

Again, the prover manages to discharge this goal by generalising the common subterm 2^n and performing an inductive proof. The binary adder program has now been defined. Brady's proof for the previous goal involves the same generalisation step and again makes use of rewriting instead of induction.

9.5.2 Inductive Families Representation: Variation

To see how robust our prover was to simple representation changes, we modified the type definitions used for the above program by arbitrarily swapping the arguments to $*$ in the definition of BitPair and swapping the arguments to + in the definition of NumCarry. The prover again successfully automated all the proofs needed. Had we defined this program with a manual proof, such as in the Idris script, such a representation change would have required the manual proof to be updated as well.

9.5.3 Subset Types Representation

To test an alternative representation, we modified the binary adder program from the previous section to capture properties using subset types instead of using inductive types. We were interested in seeing if our framework was able to support the use of both representations.

All but one of the proof obligations produced by our subset type version were successfully automated. We now briefly describe this new version of the previous program.

This time, we represent binary numbers using only simply typed inductive types. For example, reusing the same type names from before, we can define a binary number as follows:

```
Inductive Bit : Set :=
  | bit0
  | bit1.

Inductive Number : Set :=
  | none : Number
  | bit : Bit → Number → Number.
```

To capture properties of binary numbers with subset types, we make use of several functions that convert the binary number types we use above to their nat representation:

```
Fixpoint nat_of_bit (b:Bit) : nat :=
  match b with
  | bit0 ⇒ 0
  | bit1 ⇒ 1
  end.
```

```
Fixpoint num_length (n:Number) : nat :=
  match n with
  | none ⇒ 0
  | bit b m ⇒ S (num_length m)
  end.

Fixpoint nat_of_num (n:Number) : nat :=
  match n with
  | none ⇒ 0
  | bit bit0 m ⇒ nat_of_num m
  | bit bit1 m ⇒ 2 ^ (num_length m) + (nat_of_num m)
  end.
```

For example, the nat_of_num function above is used to convert a Number term into a nat term. Given a suitable implementation of addNumberAux, the following function gives a definition of addNumber, where the same properties as before are captured using subset types:

```
Program Fixpoint addNumber
  (x:Number) (y:Number | num_length y = num_length x) (carry:Bit) :
  {r:NumCarry | nat_of_nc r = nat_of_bit carry +
                              (nat_of_num x + nat_of_num y) ∧
                num_length (num_of_nc r) = num_length x} :=
  match x, y with
  | none, none ⇒ numCarry carry none
  | bit b1 t1 , none ⇒ !
  | none, bit b1 t2 ⇒ !
  | bit b1 num1, bit b2 num2 ⇒
    addNumberAux b1 b2 (addNumber num1 num2 carry)
end.
```

We found the use of subset types here cumbersome as the output types were verbose and complex. Fortunately, our testing tool alerted us to mistakes we made and the prover was able to automate all the needed proofs.

The prover was unable to automate the recursive call proof obligation generated by the addNumber function. The conclusion of this goal after destructing the result from the recursive call is as follows (the rippling annotations indicate the differences between the conclusion and the recursive call result):

$$
\begin{aligned}
&\mathsf{nat\_of\_nc}\ (\ \boxed{\mathsf{proj1\_sig}\ \ (\mathsf{addNumberAux\ b1\ b2}\ \underline{\mathsf{addNumber\_s}}\ \ )}^{\uparrow}\ )\ = \\
&\ \mathsf{nat\_of\_bit}\ \ \mathsf{carry}\ +\ (\mathsf{nat\_of\_num}\ (\ \boxed{\mathsf{bit}\ \ \mathsf{b1}\ \underline{\mathsf{num1}}}^{\uparrow}\ )\ +\ \ \mathsf{nat\_of\_num}\ (\ \boxed{\mathsf{bit}\ \ \mathsf{b2}\ \underline{\mathsf{num2}}}^{\uparrow}\ ))\ \ \wedge \\
&\mathsf{num\_length}\ (\mathsf{num\_of\_nc}\ (\ \boxed{\mathsf{proj1\_sig}\ \ (\mathsf{addNumberAux\ b1\ b2}\ \underline{\mathsf{addNumber\_s}}\ \ )}^{\uparrow}\ ))\ = \\
&\mathsf{num\_length}\ (\ \boxed{\mathsf{bit}\ \ \mathsf{b1}\ \underline{\mathsf{num1}}\ )}^{\uparrow}
\end{aligned}
$$

As the conclusion is not an equation, the rippling tactic does not have the option of weak fertilising the goal. The rippling tactic fails to strong fertilise and the proof attempt fails. However, we can help the prover succeed by destructuring the conjunction in the given, splitting the conjunction in the goal and then invoking the rippling tactic on the two subgoals produced. Rippling succeeds as it is able to weak fertilise in both subgoals as the given is an equation in each case. Clearly it would be desirable to adapt the rippling tactic to perform the manual steps described so that givens that contain conjunctions do not block rippling proofs.

9.5.4 Error Feedback

We found the error feedback helpful for the binary adder examples but with some caveats:

- We made many errors when developing the subset type version of the binary adder as we found the output types tricky to write due to the number of terms they included. The testing tool helped greatly in that it automatically told us that an error had been made. Unfortunately, as the proof obligations generated in this case study tended to contain many assumptions and a complex conclusion, the counterexample descriptions were less useful due to their verbosity. We thus found it easier to inspect the program script for errors when we were told that an error was present.

  Additionally, when we made an error writing the output type of the addNumber function, the testing tool initially did not identify the unprovable recursive call proof obligation that arose. In this case, the testing tool warned (see §8.3.2) that it could only generate test data that satisfied the side-conditions in this proof obligation approximately 1% of the time. On noticing this percentage, we manually ran the testing tool two more times before it found a counterexample. Test data was difficult to generate in this case as the tool lacks support for custom generators and the strong specification of the program being tested tightly constrained which variable instantiations were allowed.

- Our testing tool is unable to generate terms that have dependent types, such as the indexed Number type, when testing goals. So that we would get error feedback while developing the binary adder program which uses such types, whenever the automation failed, we checked for an error by manually calling the testing tool after simplifying top-level goals and discarding all assumptions. As each of the nontrivial goals simplified to an arithmetic equation that was solvable without using any of the goal assumptions, this enabled us to get feedback on when we made an error. However, as with the binary adder program written with subset types, we found the indication that an error had been made was useful but we tended not to inspect the error messages closely.

9.6 Results from Case Studies

The table in Appendix B presents a summary of how the proof automation performed in the case studies that were described in the previous sections. We ignore what we have labelled "trivial" proof obligations in the table of results. A proof obligation is labelled as trivial if it can be proven using a propositional logic decision procedure, or by reflexivity, after destructuring all subset type terms and substituting with all assumptions with the type `x = ...` for some variable `x`.

Over all of the case studies, 67 nontrivial proof obligations were generated. Out of these, 84% were successfully proven, showing that our framework offers a high degree of automation. The mean time spent on proof search was 1.60 seconds. We would consider this to be satisfactory performance for the prover to be a practical tool.

9.7 Lemma Caching Evaluation

In this section, we evaluate the impact the lemma caching feature of our prover (see §7.2) had when automating the case study proof obligations. This includes an examination of the utility of the lemma discovery feature (see §9.7.3) of our system.

9.7.1 Experimental Setup

In this experiment, we consider the following three configurations of our prover when ran against the case studies described at the start of this chapter:

DiscoverOnly: The lemmas found by lemma discovery tool are available for use by the prover. Lemmas are *not* cached after proof attempts.

CacheOnly: The lemmas found by the lemma discovery tool are *not* available for use by the prover. However, lemmas are cached after proof attempts and these lemmas can be used. The lemma cache is *not* cleared after each verification task (where the tasks are attempted in the order presented in this chapter).

BasicDefs: The prover only uses basic definitions during proofs i.e. it is not allowed to use cached lemmas.

9.7.2 Results

The prover was able to automate 47 (70%) of the goals under the BasicDefs configuration. For the DiscoverOnly and the CacheOnly configurations, the prover was able to automate 56 (84%) of the goals. The DiscoverOnly and the CacheOnly configurations succeeded on the same goals and were able to automate all the goals proven by the BasicDefs configuration. See Appendix B for the detailed results of which theorems could be automated by our prover under the various configurations.

9.7.3 Analysis

The results show that the lemma discovery tool as well as the lemma caching feature increases the proof coverage of our system. We first consider the behaviour of the lemma discovery tool.

For these case studies, the lemma discovery tool discovered lemmas concerning the +, $*$ and ++ operators. The tool conjectured and proved the following lemmas about these operators in 1.7 seconds:

$$\begin{gathered}
(x \mathbin{++} y) \mathbin{++} z = x \mathbin{++} y \mathbin{++} z \\
x * y * z = x * (y * z) \\
x * y = y * x \\
x + y + z = x + (y + z) \\
x + y = y + x
\end{gathered}$$

By comparing the results of the DiscoverOnly and the BasicDefs configuration, it can be seen that the lemmas proven during lemma discovery were required to solve 1 of the goals that arose when verifying the tail recursive factorial program (see §9.3.2) and

8 of the goals that arose when verifying the binary adder programs (see §9.5). Each of these goals was arithmetic in nature, where each proof involved induction and rippling. In each case, lemmas found during lemma discovery were required by rippling during the proof attempt to allow fertilisation to take place. For example, our prover requires more than basic definitions to discharge the following goal that arose from the tail recursive factorial case study:

$$\mathsf{n} * (\mathsf{acc} + \mathsf{p} * \mathsf{acc}) = (\mathsf{n} + \mathsf{p} * \mathsf{n}) * \mathsf{acc}$$

For the CacheOnly configuration, the results show that caching lemmas between proof attempts improved the proof coverage of the prover compared to when no lemmas were cached. For example, the CacheOnly configuration was able to automate the example goal above by using arithmetic lemmas that were cached before this goal was attempted.

It is interesting that despite the different approaches used, CacheOnly and DiscoverOnly succeeded on the same goals. We would not expect this result in general and further evaluation would be required to see how commonly this occurs. For instance, the order that the goals are attempted in will have an impact on the results of the CacheOnly configuration. Specifically, if the goal above from the tail recursive factorial case study had been attempted before any others, the proof attempt would have failed as no lemmas would have been cached at this point.

One benefit of the lemma discovery tool is that the proof coverage of the prover is less reliant on the order the goals are attempted in. However, the lemma discovery tool can only generate lemmas that have the same form as the supplied lemma templates (see §) unlike the lemma caching mechanism. Thus, scenarios must exist where a goal can be automated with a cached lemma that could not have been generated by the lemma discovery tool.

9.7.4 Summary

From this experiment, it can be seen that the lemma discovery tool and the lemma caching features of our prover increase proof coverage in practice. This agrees with previous observations that rippling-based proof automation becomes more powerful when given access to extra lemmas [Bundy et al., 1993,Dixon, 2005,Johansson, 2009]. From these results, we would consider it useful further work to consider extending our lemma discovery tool so that it is able to conjecture and prove more complex lemmas.

9.8 A Comparison with IsaPlanner

In this section, we examine how many of the proof obligations from the case studies can be automated by IsaPlanner (a tool for the Isabelle theorem prover) compared to our prover. As IsaPlanner is designed to automate inductive proofs and many of the proof obligations naturally require induction to discharge, we would expect that IsaPlanner should be able to offer some level of automation.

9.8.1 Experimental Setup

For this experiment, IsaPlanner and our prover were only be supplied with basic definitions and no additional lemmas. Recall that, using this configuration, IsaPlanner and our system gave a similar level of automation for the theorem corpus described in §7.14.

To run the experiment, we first had to translate all the case study proof obligations to Isabelle so that these could be attempted by IsaPlanner. The translation of the simply typed Coq functions that appear in the proof obligations to Isabelle was straightforward. Each proof obligation was translated to Isabelle in the following manner:

1. For each proof obligation, we first substituted any pattern matching equations that were present (see §3.3.1). This trivial simplification step is part of the default behaviour of the Program tactic and is always performed by our prover on top-level goals also.

2. The case study proof obligations contain dependently typed terms and there is no obvious way such terms can be represented in Isabelle. These features were therefore eliminated from each proof obligation before being translated to Isabelle. For each proof obligation, this was achieved by destructuring all subset type terms and discarding any assumptions concerning the inductive families used to represent binary numbers. The former step is always safe and the latter step is safe for the proof obligations from our case studies as these assumptions are not needed in any of the proofs.

For a fair comparison with IsaPlanner, our prover was ran against the case study proof obligations after the transformations described above were applied.

9.8.2 Results

Out of 67 goals, IsaPlanner was able to automate 8 (12%). In comparison, our prover was able to automate 46 of the goals (69%) while similarly configured to only use basic definitions during proofs. All of the goals discharged by IsaPlanner were also discharged by our prover. See Appendix B for the detailed results of which theorems could be automated by the two systems.

9.8.3 Analysis

IsaPlanner failed to automate a significant number of theorems and the results show that our prover is more effective at automating the proof obligations that arose from the case studies. We now describe the primary differences between the approach our prover uses to automate the proof obligations compared to the approach used by IsaPlanner:

- IsaPlanner does not take advantage of embeddings that exist in top-level goals and will perform induction when rippling could be used to guide the proof. For example, IsaPlanner fails to automate the recursive call proof obligation that arose from the example in §9.4.1.2. This proof obligation has the following form:

  $$\frac{\text{insertion\_sort\_p} \ : \ \text{length} \ \ \text{insertion\_sort\_s} \ = \text{length} \ \ \text{t}}{\text{length} \ (\boxed{\text{insert h} \ \boxed{\text{insertion\_sort\_s}}}^{\uparrow}) = \text{length} \ (\boxed{\text{h} :: \boxed{\text{t}}}^{\uparrow}}$$

  As the assumption embeds into the conclusion, rippling can be used to guide the proof. Our prover discharges this goal by rippling out the differences on the RHS of the conclusion, weak fertilising and then performing an inductive proof on the goal (see §9.4.1.2). Instead of using rippling, IsaPlanner performs induction on the top-level goal and fails to find a proof.

- IsaPlanner does not attempt to simplify or generalise goals before performing induction on the top-level goals. For example, IsaPlanner fails to automate the goal $\text{r} = 1 * \text{fact} \ \ \text{n} \rightarrow \text{r} = \text{fact} \ \ \text{n}$ (which arose from the factorial case study) with an inductive proof. Our prover automates this by simplifying the goal to $\text{fact} \ \ \text{n} + 0 = \text{fact} \ \ \text{n}$, generalising this goal to $\text{x} + 0 = \text{x}$ and then performing a simple inductive proof. IsaPlanner is however able to automate this generalised goal.

- IsaPlanner lacks reasoning techniques needed for proving impossible case proof obligations. Specifically, IsaPlanner fails to automate any of the six impossible case proof obligations that arose from the binary adder case study (see §9.5.1). For example, one of these proof obligations has the form $(\mathsf{P}{:}0 = \mathsf{S}\ x) \vdash \mathsf{False}$. Our prover discharges this goal with the trivial tactic by reasoning that the assumption has a type that is uninhabited.

- IsaPlanner fails to automate top-level goals that can be proven with only basic simplification. A simple example of this comes from the binary adder case study (see §9.5.1) where one proof obligation has the form $(\mathsf{P} : \mathsf{S}\ x = \mathsf{S}\ y) \vdash x = y$. This is solved by our prover by simplifying assumption P to $x = y$ and then using P to trivially discharge the goal. IsaPlanner instead attempts an inductive proof and fails.

9.8.4 Summary

In this experiment, it was found that our prover was able to automate significantly more of the proof obligations that arose from our case studies compared to IsaPlanner. This is in contrast to a previous experiment that compared the proof coverage of IsaPlanner and our prover on a theorem corpus (see §7.14) where both systems gave a comparable level of automation. The primary difference in the latter experiment is that almost all of the theorems from the corpus required induction to be performed directly on the top-level goals whereas this is not always the case for discharging proof obligations. IsaPlanner has primarily been designed to automate lemma statements that require induction to be performed directly on top-level goals. In contrast, our prover is designed to expect simplification, generalisation and rippling to be applicable to top-level goals. The results of this experiment suggest that the generality of IsaPlanner could be improved by adopting the strategy employed by our prover.

9.9 Answers to Research Questions

In this section, we provide answers to the research questions we proposed in §9.1 by summarising our experiences of conducting the case studies.

Which data type representations, program property representations and levels of type refinement are well supported by the proof automation?

We believe our case studies show that our framework offers broad proof automation support for programming with dependent types. In the following, we comment on several aspects of the proof automation support demonstrated:

Program properties: In the case studies, we verified the correctness of tail recursive functions, length and permutation properties of sorting functions, and the correctness of variants of a binary adder program. These program properties were described using a combination of functions that operate over lists, trees, Peano arithmetic and binary numbers. The variety of program properties and the data types used give good evidence that the automation provides generic support that will work for other types and functions that we have not yet experimented with.

Subset types: The majority of the example programs involved programming with subset types, where the propositional parts of subset types generally consisted of equational statements, the use of simple types (e.g. list and nat) and structurally recursive functions. We found the framework offered significant automation for working with this style of representation. In particular, the recursive_call, induction, ripple and generalise patterns were frequently used in the proofs required.

Inductive families: In the binary adder case study, we demonstrated that automation could be provided for working with inductive families, where the proof obligations involved had the form of non-linear arithmetic equations. Although we should examine further examples involving inductive families, we would expect support could be given whenever the type indices used produce proof obligations that involve equations, recursively defined functions and inductively defined types.

Recursion: The case studies show that the prover provides good support for working with structurally recursive functions. The quicksort example involved non-structural recursion, where partial automation was achieved. However, further work will be needed to understand how the automation copes for other examples of non-structural recursion.

Type refinement: In several examples, we changed a function that had previously returned a subset type term to one that only returned a simply typed term. For example, we did this for the helper functions in the tail recursion examples and for the insert function in the insertion sort examples. In each case, the automation provided support for the proof obligations produced, where the use of simple types typically made the proofs more challenging.

Robustness to change: In each set of case studies, we examined programs that were created by making slight changes to previously written programs to see how the automation coped. For example, in the tail recursion and binary adder examples, we changed the order of certain arguments in such a way that the program behaviour and the property being verified were the same but the proofs required were different. In each case, the automation was robust to these small changes. This gives further evidence of the generality of the tactics. Moreover, this support is useful when refactoring programs as, without proof automation, small representation changes usually require that we update manual proofs.

As seen in the quicksort and tree sort examples, the prover is unable to automate any proofs that required piecewise fertilisation. Additionally, the termination proofs for the quick sort examples were problematic as the prover does not support the use of inductive predicates yet.

How often can proof automation failures be overcome with user hints? What level of expertise is required to give effective hints?

On several occasions, we managed to give hints to help the prover discharge a problematic proof obligation. In the quicksort case study, we supplied a wave rule hint to unblock a rippling proof. In the tree sort case study, we supplied a generalisation hint to work around a case where the generalisation tactic was overgeneralising a goal. We personally found formulating these particular hints intuitive, but we are aware that a good understanding of what each tactic does during proof search is needed for this to be a realistic option in many cases. For example, some of the hints we gave would require the user to understand rippling and rippling annotations. We note that, in cases where the proof was blocked because of the lack of support for piecewise fertilisation and inductive predicates, there was no obvious way we could give a hint to help the prover.

How helpful were the error feedback facilities during development?

At the end of each case study, we commented on the usefulness of the error feedback feature of our framework. In general, we felt the feedback greatly improved our ability to identify and fix errors. For example, we believe that the binary adder case study would have been much more difficult to complete without this since we made numerous errors that were caught by our testing tool.

However, we did note that when capturing the permutation properties of the sorting algorithms, the testing tool was less useful as it was unable to test goals that included universally quantified assumptions. Additionally, when writing the binary adder with the use of subset type, we noted that, although the indication that there was an error was helpful, many of the actual error messages were difficult to interpret.

9.10 Related Work in Dependently Typed Programming Environments

We now compare the facilities offered by our framework, which provides integrated proof automation and error feedback, to the support provided by other dependently typed programming environments. We consider Agda first, which has both an inductive proof automation tool and a testing tool available for it.

The Agsy proof automation tool for Agda [Lindblad and Benke, 2006] 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 [Lindblad and Benke, 2006, §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 to discover simple lemmas that are needed during some proofs [Lindblad and Benke, 2006, §4]. It is difficult to make a more formal comparison between our system and Agsy as we are unable to obtain a version of Agsy to perform experiments with and there is no corpus available that documents which theorems Agsy can automate.

Although Agsy does not currently cache and reuse the proofs it finds, delayed generalisation could be implemented similarly in Adga. An interesting difference with our work is that the Agsy tool directly inserts the terms it constructs into the program script. For this reason, Agsy contains features for improving the readability of the terms found, such as searching for short proofs and naming variables in a readable manner.

The testing tool for Agda [Qiao Haiyan, 2003] (which we mentioned in §8.4) is similar to ours in that it uses a QuickCheck-like approach and testing occurs within the target language. However, Agda's testing tool is more powerful than ours in that it supports generators for inductive families, generators for functions and custom generators. For example, this support for inductive families would have been useful for finding counterexamples to the proof obligations that arose in our binary adder case study.

We note that Agsy and Agda's testing tool are not integrated in any fashion which is in contrast to our framework where testing is integrated to provide error feedback, feedback for faulty hints and is used to guide proof search. However, Agda's testing tool has been used in combination with interactive theorem proving and a boolean formula model checker to conduct program verification case studies [Qiao Haiyan, 2003].

Outside of Agda, we are not aware of any environment intended for dependently typed programming that includes inductive proof automation. For example, all of our case studies would require manual proofs if written in Sage, Epigram, ATS or Idris, although we note that Sage and ATS provide automation for linear arithmetic goals.

Epigram, ATS and Idris also lack facilities for identifying errors and providing error feedback compared to what our framework offers. However, Sage does makes use of counterexamples as a way to detect errors. If Sage cannot verify a program property statically, a dynamic check is added to the compiled program that checks the property at run-time. If this run-time check is violated, the user is presented with a corresponding counterexample. The counterexample is stored in a database so that future proof obligations of the same form can be rejected at compile time.

9.11 Related Work in Inductive Proof Automation

We are unaware of any tactics in Coq that can automate theorems 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 [Barthe et al., 2006]. Moreover, Coq's auto tactic [Bertot and Castéran, 2004], 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.

The Boyer-Moore theorem prover [Boyer and Moore, 1979], and its successor ACL2 [Kaufmann and Moore, 1997], are well known for their inductive proof automation and are likely to be able to automate theorems similar to those that we have presented. ACL2 features a complex and fine-tuned simplification tactic which is used to simplify step case proofs, in contrast to our approach based on rippling. ACL2 also uses heuristics for choosing appropriate induction principles and induction variables. At the moment, our prover only supports proofs that use standard induction principles where this can limit the kinds of inductive proofs rippling can automate [Johansson, 2009, §5.6]. As with our work, when the automation fails in ACL2, the user has the option of providing hints. ACL2's hint feature is more advanced that ours in that many more options are available. For example, the user can instruct the prover to apply a lemma to a specific subgoal in the proof attempt and dictate which induction principle to use.

Rippling has been implemented in other systems, such as in Clam [Bundy et al., 1990], NuPrl [Pientka and Kreitz, 1998a] and IsaPlanner [Dixon, 2005]. IsaPlanner uses rippling to automate inductive proofs in a simply typed setting within a proof planning framework. Like our prover, IsaPlanner includes support for rippling proofs that involve case splits and multiple hypotheses [Johansson, 2009]. An important difference between IsaPlanner and our prover is that we always attempt rippling, simplification and generalisation on a top-level goal before an inductive proof is considered. In contrast, IsaPlanner always attempts to discharge top-level goals by performing induction first. A comparison of the level of automation offered by our system and IsaPlanner can be found in §7.14 and §9.8.

9.12 Conclusions

In our case studies, we have shown that our framework provides practical support for developing dependently typed programs. These case studies involved verifying tail recursive functions, sorting functions, and a binary adder. A variety of data types were used and different program properties were captured to demonstrate the generality of our approach and framework. The proof automation was found to discharge a significant amount of the proof obligations produced. Moreover, we reported that we found the error feedback and hinting facilities of our framework useful in practice during our case studies.

Chapter 10

Conclusions and Further Work

We now give a review of the contributions of the thesis and then discuss whether the hypothesis presented in the first chapter has been verified. This is followed with some suggestions on areas for future research.

10.1 Framework Overview

In this thesis, we have presented a framework designed to make dependently typed functional programming with user-defined properties easier. The primary features of this framework are as follows:

- Testing is employed to identify and give feedback to errors that are indicated by unprovable proof obligations (see Chapter 8).
- Proof automation that supports reasoning about inductively defined types and recursively defined functions is employed to discharge proof obligations (see chapters 6 and 7).
- To increase proof coverage, lemmas found during proof search are cached for reuse in future proof attempts (see §7.2).
- Should the automation fail to discharge a goal, a trace of the proof attempt is given and the user has the opportunity to help the prover find a proof by giving high-level hints in the form of lemma conjectures (see §7.13). Testing is also used here to give feedback to faulty hints.

The two main components of our framework are as follows:

- The *proof automation* component is composed of several tactics that are structured using the Boyer-Moore waterfall approach [Boyer and Moore, 1979]. Inductive proof automation is provided, primarily with the use of tactics that perform simplification, rippling [Bundy et al., 2005] and generalisation [Aubin, 1976, Boyer and Moore, 1979, Aderhold, 2007]. For the latter, the testing component is employed to identify overgeneralisations.

- The *testing tool* component uses a QuickCheck-like [Claessen and Hughes, 2000] approach to identify unprovable goals, where the counterexamples found are used for providing error feedback.

We believe that this framework is effective at making development more practical as it gives integrated support for several common, and frequently challenging, activities that take place when programming with dependent types: identifying errors, fixing errors and constructing proofs. The generic nature of the framework means support can be given for capturing a wide range of program properties and the user is not restricted to working with predefined definitions. Further to this, a benefit of our approach is that the modular architecture of the proof automation gives a foundation that can be built upon for addressing new domains. For example, to give support for further program properties, new tactics could be introduced to various stages of the waterfall and existing tactics could be extended.

In DML [Xi, 1998], the program properties the user can capture are restricted to those that concern linear arithmetic, but the proof automation provided is decidable. With our system, we can support many more forms of program properties but, with this freedom, it is harder to make guarantees to the user about what proofs can be automated. The hinting feature was added so that users can sometimes avoid having to resort to manual proofs on the occasions where the automation fails. However, the user has to have some knowledge of theorem proving and how the proof automation works to take advantage of this. Likewise, error feedback is only available when the user works with properties that can be tested.

10.2 Contributions

The combination of features and their integration within this framework is novel compared to what is available in current dependently typed programming environments. We note that the underlying ideas and approach used could be equally applied to give support in other languages with dependent types, such as ATS, Epigram, Agda and Idris. In particular, we have shown how to provide effective proof automation for supporting program properties that involve inductively defined types and recursively defined functions.

As a by-product of developing a prototype of our framework, we have introduced inductive proof automation and a QuickCheck-like testing tool to Coq. As far as we know, there are no existing tools that provide similar capabilities in this system. As such, we believe this contribution can be of practical value to the Coq community.

Moreover, aspects of our work concerning inductive proof automation have applications elsewhere:

- We devised delayed generalisation as a technique for identifying irrelevant assumptions at the end of a proof for the purpose of eliminating the corresponding irrelevant subformulae from cached lemmas (see §7.10). This idea could also be applied in other provers that cache lemmas, such as in IsaPlanner [Dixon, 2005].

- We introduced heuristics that can automatically identify, from a collection of lemmas, a terminating set of rewrite rules suitable for simplification (see §7.11). These heuristics could also be employed by IsaPlanner to make more productive use of cached lemmas, including those found by IsaPlanner's lemma discovery tool IsaCoSy [Dixon, 2005,Johansson, 2009].

- We have given further evidence that the rippling technique can be productively applied outside of its traditional use in proof planning. In most presentations, the utility of rippling is demonstrated as part of proof planning [Bundy, 1988,Dixon, 2005]. In contrast, our proof automation is structured using a Boyer-Moore style waterfall, where this waterfall includes a call to a rippling tactic. Rippling has also been used without proof planning in Nuprl [Pientka and Kreitz, 1998a].

10.3 Hypothesis

We now consider the evidence we have produced for the hypothesis presented in the first chapter. The hypothesis was as follows:

"This framework makes dependently typed programming significantly easier"

We believe that this has been shown with the evidence from our case studies where we reported on our experiences developing dependently typed programs with the help of our framework (see Chapter 9). In these case studies, we verified tail recursive functions, sorting functions, and variations of a binary adder. This included the use of programs based on examples found in ATS, Coq and Idris respectively.

The programs that we wrote made use of a variety of data types, program properties and various levels of type refinement. Program properties were described using inductively defined types and recursively defined functions, generally with the use of equality statements. The majority of our examples made use of subset types, although the binary adder case study involved the use of inductive families.

A significant amount of the proof obligations generated (84%) were discharged automatically, thereby providing good evidence that the prover is effective in practice. Moreover, the use of example programs that only differed by small representation changes demonstrated the robustness of the proof automation.

The high level of automation relieved the burden of writing a large portion of the proofs by hand, a fact that undoubtedly made development significantly easier. Moreover, in several cases where the proof automation failed, we found that we were able to use the lemma hinting facilities to help the prover find a proof. We also reported that the error feedback provided by our framework identified many errors for us in practice and that the error messages given were usually helpful in suggesting what changes to make. However, we did note that the error messages were sometimes hard to interpret when we were capturing weak specifications and when the proof obligations generated were complex.

10.4 Further Work

We now describe several areas of future research into providing better support for programming with dependent types. The first topics we cover concern providing more support for inductive families and non-structural recursion, as well as integrating domain specific techniques into our prover. From the work done so far, it is not entirely clear what extensions would need to be added to our automation to support these. We then cover topics where it is comparatively easier to know what work needs to be done next. For example, we can look to previous work to incorporate extensions to give automation for proofs that involve piecewise fertilisation and existential quantifiers.

We would consider the topics of adding support for inductive families and inductive predicates some of the most important for making dependently typed programming more practical as these such representations appear often in most Coq, Agda and

Epigram programs. Moreover, the integration of domain specific techniques into our prover is likely to be important for scaling to more complex program verification tasks.

10.4.1 Inductive Families

The majority of the dependently typed programs we have considered so far capture program properties using subset types as opposed to dependently typed inductive families. For future work, we would want to give further support for the latter. This would require looking at more varied and more complex examples of the proof patterns that arise when programming with inductive families. For example, we could consider inductive families for representing ordered lists [Altenkirch et al., 2005] and list permutations [Brady et al., 2008]. Sozeau's formalisation of finger trees would be a challenging case study to consider as this makes extensive use of inductive families in combination with inductive predicates and subset types [Sozeau, 2007a].

For providing error feedback for the above, we would need to extend our testing tool with term generators for inductive families. Agda's testing tool can test goals that include inductive families as long as the user writes custom generators for the inductive families used [Qiao Haiyan, 2003]. For a practical testing tool however, it would be desirable to minimise the need for the user to have to write the generators themselves.

10.4.2 Integrating Domain Specific Tools and Libraries

Our automation work has focused on constructing proofs about user-defined properties with the use of induction. Typical dependently typed programs will make use of user-defined types in combination with common standard definitions, such as simply typed lists and Peano arithmetic. Useful further work would involve integrating domain specific tactics into our automation along with methods for making use of theorems from existing libraries. These topics will be important for creating a tool that scales to larger and more complex programs than those which we have looked at so far.

10.4.3 Non-Structural Recursion

The majority of the example programs we have considered so far have involved structural recursion. If we wish to extend our proof automation support to non-structurally recursive functions, further examples will need to be examined to determine what extensions would be required. Although we showed that our framework gave some automated support for the proof obligations that arose in the case study involving quicksort (which was defined using non-structural recursion) in §9.4.3, it is unlikely that other examples will be as straightforward.

10.4.4 Inductive Predicates

Currently, we do not provide support for working with user-defined properties that involve inductive predicates. The latter are used heavily in many dependently typed programs and support for inductive predicates would make our framework even more practical.

Our rippling tactic would need several extensions to support inductive predicates. Given that $\mathsf{even\ n}$ is an inductive predicate, we would need to allow the use of rules such as $\forall\ \mathsf{n\ m},\ \mathsf{even\ n} \rightarrow \mathsf{even\ (n * m)}$ and $\forall\ \mathsf{n},\ \mathsf{even\ n} \leftrightarrow\ \mathsf{even\ (S\ (S\ n))}$ for rippling proof steps. Furthermore, we would need to extend the weak fertilisation step to allow the use of user-defined relations. The recently improved rewriting support for working with arbitrarily relations in Coq is likely to make implementing these extensions easier [Sozeau, 2009]. For our simplification tactic, we would need additional heuristics for simplifying goals that contain inductive predicates, which would likely include the use of inversion [Cornes and Terrasse, 1995].

Our testing tool would need to be extended to test goals that contain user-defined inductive predicates. Agda's testing tool shows how Prolog-like search can be used to test goals that include user-defined inductive predicates in limited cases [Qiao Haiyan, 2003]. In other situations, testing support could be given by asking the user to supply a mapping between an inductive predicate and an equivalent function so that the goal can be transformed to a testable form.

10.4.5 Infinite Data Structures

An interesting extension would be to provide support for writing dependently typed programs that involve infinite data types, such as lazy lists. Such types can be defined in Coq using coinduction [Bertot, 2005]. Of particular relevance to extending our prover to support coinductive proofs, we are aware that work exists on coinductive proof automation in Clam [Dennis, 1998]. For providing error feedback, we would need to extend the generate and test phases of our testing tool to cope with the inclusion of infinite data types.

10.4.6 Piecewise Fertilisation

We do not currently support rippling proofs where a given includes an implication. For example, during our case studies, we required an inductive proof of the following theorem to show that an implementation of quicksort would terminate (see §9.4.3.2):

```
∀ x y z, list_perm (x ++ y) z → length x < S (length z)
```

The step case of this inductive proof requires rippling with a given that contains an implication. Likewise, when the propositional part of a subset type contains an implication, similar reasoning will be required to discharge recursive call proof obligations. Proofs such as these could be supported by extending rippling to perform piecewise fertilisation [Armando et al., 1999]. IsaPlanner is similarly unable to automate these kinds of proofs at the moment [Johansson, 2009], but there are plans to add to the necessary extensions in the future [Dennis and Dixon, 2009].

10.4.7 Improved Error Feedback

The error messages discussed in Chapter 8 describe to the user how a proof obligation generated by a faulty program is unprovable with the use of a counterexample. We noted that the error messages could be more helpful when weakly specified functions were used (see §8.1.3) and we found in the binary adder case study (see §9.5) that the error messages produced were tricky to interpret as the proof obligations contained numerous assumptions and complex terms.

To produce more helpful error messages, we have considered generating error messages that make direct use of the top-level function being defined. To describe this idea by example, the counterexample to the proof obligation shown in §8.1 could be presented in the following manner:

```
Given the following output from "intersperse":

output = intersperse x y
       = intersperse [1] []
       = [1]

The propositional part of the output type of
"intersperse" is uninhabitable:
```

```
length output = (length x)   * (length y)
   length [1] = (length [1]) * (length [])
            1 = 1 * 0
            1 = 0
```

We think this style of error message is likely to be easier to understand and more concise in many cases compared to examining counterexamples to proof obligations.

10.4.8 Existential Quantifiers

We have yet to consider support for proof obligations that contain existential quantifiers. Of relevance, rippling has been applied in Nuprl to automate proofs that contain existential quantifiers [Pientka and Kreitz, 1998a] and such an extension is likely to be useful to our proof automation. For testing support, we are aware that SmallCheck has some support for testing existentially quantified conjectures [Runciman et al., 2008].

10.5 Summary

In this thesis, we presented a framework that combines proof automation and testing for the purpose of supporting dependently typed programming. In this concluding chapter, we described the contributions of the thesis and outlined the evidence for our hypothesis that the framework presented makes dependently typed programming significantly easier. We then discussed possible further research that we believe can be used to make programming with dependent types more practical in future.

Bibliography

[Aczel, 1977] Aczel, P. (1977). An introduction to inductive definitions. In Barwise, J., editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland.

[Adams and Dennis, 2003] Adams, A. A. and Dennis, L. A. (2003). Rippling in PVS. In Archer, M., Vito, B. D., and Munoz, C., editors, *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, pages 84–91. NASA Technical Report CP-2003-212448.

[Aderhold, 2007] Aderhold, M. (2007). Improvements in formula generalization. In Pfenning, F., editor, *Automated Deduction - CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 231–246. Springer.

[Allen et al., 2000] Allen, S. F., Constable, R. L., Eaton, R., Kreitz, C., and Lorigo, L. (2000). The Nuprl open logical environment. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 170–176, London, UK. Springer-Verlag.

[Altenkirch et al., 2005] Altenkirch, T., McBride, C., and McKinna, J. (2005). Why dependent types matter. `http://www.e-pig.org/downloads/ydtm.pdf`.

[Armando et al., 1999] Armando, A., Smaill, A., and Green, I. (1999). Automatic synthesis of recursive programs: The proof-planning paradigm. *Autom. Softw. Eng*, 6(4):329–356.

[Aspinall et al., 2008] Aspinall, D., Denney, E., and Lüth, C. (2008). A tactic language for hiproofs. In Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., and Wiedijk, F., editors, *AISC/MKM/Calculemus*, volume 5144 of *Lecture Notes in Computer Science*, pages 339–354. Springer.

[Aubin, 1976] Aubin, R. (1976). *Mechanizing structural induction*. PhD thesis, The University of Edinburgh.

[Augustsson, 1984] Augustsson, L. (1984). A compiler for Lazy ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227, New York, NY, USA. ACM Press.

[Augustsson, 1998] Augustsson, L. (1998). Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250.

[Augustsson and Carlsson, 1999] Augustsson, L. and Carlsson, M. (1999). An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*.

[Barnett et al., 2005] Barnett, M., Leino, K. R. M., and Schulte, W. (2005). The Spec# programming system: an overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., and Muntean, T., editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag.

[Barthe et al., 2006] Barthe, G., Forest, J., Pichardie, D., and Rusu, V. (2006). Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In *Proceedings of 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag.

[Basin and Walsh, 1996] Basin, D. A. and Walsh, T. (1996). A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180.

[Bertot, 2005] Bertot, Y. (2005). Coinduction in Coq. In *Lecture Notes of TYPES Summer School 2005, Sweden, Volume II*.

[Bertot, 2008] Bertot, Y. (2008). Coq in a hurry. `http://arxiv.org/abs/cs/0603118`.

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

[Bertot and Théry, 2008] Bertot, Y. and Théry, L. (2008). Dependent types, theorem proving, and applications for a verifying compiler. In *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005*, pages 173–181, Berlin, Heidelberg. Springer-Verlag.

[Blanchette and Nipkow, 2009] Blanchette, J. C. and Nipkow, T. (2009). Nitpick: A counterexample generator for higher-order logic based on a relational model finder. Technical report, In Tests and Proofs 2009: Short Papers, ETH.

[Boulton, 1993] Boulton, R. J. (1993). Boyer-Moore Automation for the HOL System. In *HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 133–142. North-Holland/Elsevier.

[Bove and Capretta, 2005] Bove, A. and Capretta, V. (2005). Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708. Cambridge University Press.

[Bove et al., 2009] Bove, A., Dybjer, P., and Norell, U. (2009). A brief overview of Agda - a functional language with dependent types. In *TPHOLs, 22nd International Conference, LNCS 5674*, pages 73–78.

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

[Boyer and Moore, 1988] Boyer, R. S. and Moore, J. S. (1988). Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. *Machine intelligence*, 11:83–124.

[Brady, 2007] Brady, E. (2007). Ivor, a proof engine. In *Proceedings of Implementation of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*. Springer.

[Brady, 2008] Brady, E. (2008). Idris, a language with dependent types. In *IFL 2008*.

[Brady et al., 2008] Brady, E., Herrmann, C., and Hammond, K. (2008). Lightweight invariants with full dependent types. In *Proceedings of TFP 2008*.

[Brady, 2005] Brady, E. C. (2005). *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University.

[Bundy, 1988] Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 111–120, London, UK. Springer-Verlag.

[Bundy, 2001] Bundy, A. (2001). The automation of proof by mathematical induction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science.

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

[Bundy et al., 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: a heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2):185–253.

[Bundy et al., 1990] Bundy, A., van Harmelen, F., Horn, C., and Smaill, A. (1990). The Oyster-Clam System. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK. Springer-Verlag.

[Burton, 1982] Burton, F. W. (1982). An efficient functional implementation of FIFO queues. *Inf. Process. Lett.*, 14(5):205–206.

[Calcagno et al., 2003] Calcagno, C., Taha, W., Huang, L., and Leroy, X. (2003). Implementing multi-stage languages using ASTs, GenSym, and Reflection. In *In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, Generative Programming and Component Engineering (GPCE), Lecture Notes in Computer Science*, pages 57–76. Springer-Verlag.

[Cardelli, 1994] Cardelli, L. (1994). The Quest language and system.

[Carlier and Dubois, 2008] Carlier, M. and Dubois, C. (2008). Functional testing in the Focal environment. In *Tests and Proofs*, pages 84–98.

[Chen and Xi, 2005] Chen, C. and Xi, H. (2005). Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia.

[Chlipala, 2007] Chlipala, A. J. (2007). Position paper: Thoughts on programming with proof assistants. *Electr. Notes Theor. Comput. Sci*, 174(7):17–21.

[Christian, 1993] Christian, J. (1993). Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10:95–113. 10.1007/BF00881866.

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68.

[Church, 1941] Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press.

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y. ACM Press.

[Colton and Pease, 2005] Colton, S. and Pease, A. (2005). The TM System for Repairing Non-Theorems. *Electronic Notes in Theoretical Computer Science*, 125(3):87–101.

[Constable et al., 1986] 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., and Smith, S. F. (1986). *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ.

[Coq development team, 2006] Coq development team (2006). *The Coq proof assistant reference manual*. LogiCal Project. Version 8.1.

[Coquand, 1998] Coquand, C. (1998). The AGDA proof system homepage. `http://www.cs.chalmers.se/~catarina/agda/`.

[Cornes and Terrasse, 1995] Cornes, C. and Terrasse, D. (1995). Automating inversion of inductive predicates in Coq. In *TYPES*, pages 85–104.

[Cui et al., 2005] Cui, S., Donnelly, K., and Xi, H. (2005). ATS: A language that combines programming with theorem proving. In *Lecture Notes in Computer Science*, pages 310–320. Springer.

[Dantzig and Eaves, 1973] Dantzig, G. B. and Eaves, B. C. (1973). Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory, Series. A*, 14(3):288–297.

[de Bruijn, 1980] de Bruijn, N. (1980). A survey of the project AUTOMATH. In Seldin, J. P. and Hindley, J. R., editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press.

[Delahaye, 2000] Delahaye, D. (2000). A Tactic Language for the System Coq. In Parigot, M. and Voronkov, A., editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 85–95, Reunion Island (France). Springer.

[Denney, 2001] Denney, E. (2001). The synthesis of a Java card tokenization algorithm. In *Automated Software Engineering*, pages 43–50. IEEE Computer Society.

[Denney et al., 2006] Denney, E., Power, J., and Tourlas, K. (2006). Hiproofs: A hierarchical notion of proof tree. *Electronic Notes in Theoretical Computer Science*, 155:341–359.

[Dennis, 1998] Dennis, L. A. (1998). *Proof Planning Coinduction*. PhD thesis, Edinburgh University.

[Dennis and Dixon, 2009] Dennis, L. A. and Dixon, L. (2009). Adapting piecewise fertilisation to reason about hypotheses. In Hustadt, U., editor, *Proceedings of the Automated Reasoning Workshop 2009*.

[Dennis and Nogueira, 2005] Dennis, L. A. and Nogueira, P. (2005). What can be learned from failed proofs of non-theorems? In Hurd, J., Smith, E., and Darbari, A., editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005): Emerging Trends Proceedings*, pages 45–58. Technical Report PRG-RP-05-2, Oxford University Computer Laboratory.

[Dennis and Smaill, 2001] Dennis, L. A. and Smaill, A. (2001). Ordinal arithmetic: A case study for rippling in a higher order domain. In *TPHOLs '01: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 185–200, London, UK. Springer-Verlag.

[Detlefs et al., 2005] Detlefs, D., Nelson, G., and Saxe, J. B. (2005). Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473.

[Dixon, 2005] Dixon, L. (2005). *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh.

[Dixon and Fleuriot, 2003] Dixon, L. and Fleuriot, J. D. (2003). IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283.

[Dixon and Fleuriot, 2004] Dixon, L. and Fleuriot, J. D. (2004). Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98.

[Dybjer, 1991] Dybjer, P. (1991). Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press.

[Dybjer, 1994] Dybjer, P. (1994). Inductive families. *Formal Asp. Comput.*, 6(4):440–465.

[Dybjer et al., 2003a] Dybjer, P., Haiyan, Q., and Takeyama, M. (2003a). Combining testing and proving in dependent type theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 188–203. SpringerVerlag.

[Dybjer et al., 2003b] Dybjer, P., Haiyan, Q., and Takeyama, M. (2003b). Verifying Haskell programs by combining testing and proving. *Quality Software, International Conference on*, 0:272.

[Fogarty and Pasalic, 2007] Fogarty, S. and Pasalic, E. (2007). Concoqtion: indexed types now. In *In Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 112–121. ACM Press. ISBN.

[Giménez and Castéran, 2005] Giménez, E. and Castéran, P. (2005). A tutorial on [co-]inductive types in Coq. available at `http://coq.inria.fr/doc`.

[Gonthier, 2007] Gonthier, G. (2007). The Four Colour Theorem: Engineering of a formal proof. In Kapur, D., editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer.

[Gow, 2004] Gow, J. (2004). *The Dynamic Creation of Induction Rules Using Proof Planning*. PhD thesis, School of Informatics, University of Edinburgh.

[Griffioen and Huisman, 1998] Griffioen, D. and Huisman, M. (1998). A comparison of PVS and Isabelle/HOL. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 123–142, London, UK. Springer-Verlag.

[Grobauer, 2001] Grobauer, B. (2001). Cost recurrences for DML programs. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 253–264, New York, NY, USA. ACM Press.

[Gronski et al., 2006] Gronski, J., Knowles, K., Tomb, A., Freund, S. N., and Flanagan, C. (2006). Sage: Hybrid checking for flexible specifications. In *In Scheme and Functional Programming Workshop*, pages 93–104.

[Hallgren, 1998] Hallgren, T. (1998). The proof editor Alfa. `http://www.cs.chalmers.se/~hallgren/Alfa/`.

[Howard, 1980] Howard, W. (1980). The formulas-as-types notion of construction. In Seldin, J. P. and Hindley, J. R., editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, New York, NY.

[Hudak et al., 1992] Hudak, P., Jones, S. P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. (1992). Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164.

[Hurd, 2001] Hurd, J. (2001). Predicate subtyping with predicate sets. In *TPHOLs*, pages 265–280.

[Hutter and Sengler, 1996] Hutter, D. and Sengler, C. (1996). INKA: The next generation. In McRobbie, M. A. and Slaney, J. K., editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 288–292. Springer.

[IBM, 1954] IBM (1954). Specifications for the IBM Mathematical FORmula TRANSlating system. Preliminary report, IBM Corp., Programming Research Group, Applied Sciences Division, New York, NY, USA.

[Ireland, 1992] Ireland, A. (1992). The use of planning critics in mechanizing inductive proofs. In *Logic Programming and Automated Reasoning*, pages 178–189.

[Ireland, 1995] Ireland, A. (1995). Rippling to meet the challenge. Edinburgh Dream Group Blue Book Note 1049.

[Ireland and Bundy, 1996] Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111.

[Johansson, 2009] Johansson, M. (2009). *Automated Discovery of Inductive Lemmas*. PhD thesis, University of Edinburgh.

[Johansson et al., 2006] Johansson, M., Bundy, A., and Dixon, L. (2006). Best-first rippling. In Stock, O. and Schaerf, M., editors, *Reasoning, Action and Interaction in AI Theories and Systems*, volume 4155 of *Lecture Notes in Computer Science*, pages 83–100. Springer.

[Kammüller, 2000] Kammüller, F. (2000). Modular reasoning in Isabelle. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 99–114, London, UK. Springer-Verlag.

[Kaufmann and Moore, 1997] Kaufmann, M. and Moore, J. S. (1997). An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213.

[Koopman et al., 2002] Koopman, P., Alimarine, A., Tretmans, J., and Plasmeijer, R. (2002). Gast: Generic automated software testing. In *The 14th International Workshop on the Implementation of Functional Languages, IFL02, Selected Papers, volume 2670 of LNCS*, pages 84–100. Springer.

[Kreisel, 1965] Kreisel, G. (1965). Mathematical logic. In Saaty, T. L., editor, *Lectures on Modern Mathematics*, page 95. John Wiley and Sons, New York.

[Leino et al., 2000] Leino, K. R. M., Nelson, G., and Saxe, J. B. (2000). ESC/Java user's manual. Technical note, Compaq Systems Research Center.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115.

[Lindblad and Benke, 2006] Lindblad, F. and Benke, M. (2006). A tool for automated theorem proving in Agda. *Lecture Notes in Computer Science*, 3839/2006:154–169.

[Loader, 1998] Loader, R. (1998). Notes on simply typed lambda calculus. Technical report, The University of Edinburgh. Report number ECS-LFCS-98-381.

[Luo, 1994] Luo, Z. (1994). *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press.

[Magaud, 2003] Magaud, N. (2003). Programming with dependent types in Coq: a study of square matrices. `http://dpt-info.u-strasbg.fr/~magaud/UNSW/Coq/Matrices/`.

[Magnusson and Nordström, 1994] Magnusson, L. and Nordström, B. (1994). The ALF proof editor and its proof engine. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 213–237, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

[Martin-Löf, 1971] Martin-Löf, P. (1971). A theory of types. Manuscript.

[McBride, 2000] McBride, C. (2000). *Dependently Typed Functional Programs and their Proofs*. PhD thesis, LFCS, University of Edinburgh, Edinburgh, Scotland.

[McBride, 2004] McBride, C. (2004). Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170.

[McBride, 2005] McBride, C. (2005). The Epigram prototype: a nod and two winks.

[McBride and McKinna, 2004] McBride, C. and McKinna, J. (2004). The view from the left. *Journal of Functional Programing*, 14(1):69–111.

[McCaslan et al., 2007] McCaslan, R., Bundy, A., and Autexier, S. (2007). Automated discovery of inductive theorems. In Zalewska, R. M. A., editor, *From insight to proof - Jubilee Book for Andrzej Trybulec*, volume 10 (23), pages 135–150. University of Bialystok.

[McCune, 2001] McCune, W. (2001). MACE 2.0 reference manual and guide. `http://arxiv.org/abs/cs/0106042`.

[Mckinna and Brady, 2005] Mckinna, J. and Brady, E. (2005). Phase distinctions in the compilation of Epigram.

[McKinna and Wright, 2006] McKinna, J. and Wright, J. (2006). A type-correct, stack-safe, provably correct, expression compiler in Epigram.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

[Milner et al., 1997] Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[Murthy, 1990] Murthy, C. R. (1990). *Extracting constructive content from classical proofs*. PhD thesis, Cornell University, Ithaca, NY, USA.

[Nash, 2000] Nash, J. C. (2000). The (Dantzig) simplex method for linear programming. *Computing in Science and Eng.*, 2(1):29–31.

[Necula, 1997] Necula, G. C. (1997). Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris.

[Nipkow, 2004] Nipkow, S. B. T. (2004). Random testing in Isabelle/HOL. In Cuellar, J. and Liu, Z., editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

[Norell, 2007] Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

[Okasaki, 1998] Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press, Cambridge, England.

[Owre, 2006] Owre, S. (2006). Random testing in PVS. In *Workshop on Automated Formal Methods*.

[Owre et al., 1999] Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. (1999). *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA.

[Papapanagiotou, 2007] Papapanagiotou, P. (2007). On the automation of inductive proofs in HOL light. Master Thesis, University of Edinburgh.

[Parent, 1995] Parent, C. (1995). Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *MPC*, pages 351–379.

[Paulin-Mohring, 1989] Paulin-Mohring, C. (1989). *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7.

[Paulin-Mohring, 1993] Paulin-Mohring, C. (1993). Inductive definitions in the system Coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK. Springer-Verlag.

[Paulson, 1991] Paulson, L. C. (1991). Isabelle system for constructive type theory. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/library/CTT/.

[Pfenning, 1993] Pfenning, F. (1993). Refinement types for logical frameworks. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299.

[Pientka and Kreitz, 1998a] Pientka, B. and Kreitz, C. (1998a). Automating inductive specification proofs in Nuprl. *Fundamenta Informaticae*, 1(2):189 – 209.

[Pientka and Kreitz, 1998b] Pientka, B. and Kreitz, C. (1998b). Instantiation of existentially quantified variables in inductive specification proofs. In *AISC '98: Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation*, pages 247–258, London, UK. Springer-Verlag.

[Pierce, 2002] Pierce, B. C. (2002). *Types and programming languages*. MIT Press, Cambridge, MA, USA.

[Pollack, 1994] Pollack, R. (1994). *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh.

[Qiao Haiyan, 2003] Qiao Haiyan (2003). *Testing and Proving in Dependent Type Theory*. PhD thesis, School of Computer Science and Engineering, Chalmers University of Technology.

[Runciman et al., 2008] Runciman, C., Naylor, M., and Lindblad, F. (2008). Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48, New York, NY, USA. ACM.

[Shankar and Owre, 1999] Shankar, N. and Owre, S. (1999). Principles and pragmatics of subtyping in PVS. In Bert, D., Choppy, C., and Mosses, P., editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France. Springer-Verlag.

[Slaney, 1994] Slaney, J. (1994). FINDER: Finite domain enumerator system description. In Bundy, A., editor, *Automated Deduction-CADE-12*, pages 798–801. Springer, Berlin, Heidelberg.

[Slind et al., 1998] Slind, K., Gordon, M., Boulton, R., and Bundy, A. (1998). System description: An interface between CLAM and HOL. In Kirchner, C. and Kirchner, H., editors, *Proceedings of the Fifteenth International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Lindau, Germany. Springer.

[Smaill and Green, 1996] Smaill, A. and Green, I. (1996). Higher-order annotated terms for proof search. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 399–413, London, UK. Springer-Verlag.

[Sozeau, 2007a] Sozeau, M. (2007a). Program-ing Finger Trees in Coq. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 13–24. ACM Press.

[Sozeau, 2007b] Sozeau, M. (2007b). Subset coercions in Coq. In *TYPES'06*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer.

[Sozeau, 2008] Sozeau, M. (2008). *Un environnement pour la programmation avec types dependants*. Thèse de doctorat, Université Paris-Sud.

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

[van Deursen et al., 1993] van Deursen, A., Klint, P., and Tip, F. (1993). Origin tracking. *Journal of Symbolic Computation*, 15(5-6):523–545.

[Walther, 1994] Walther, C. (1994). Mathematical induction. In *Handbook of logic in artificial intelligence and logic programming*, pages 127–228. Oxford University Press, Inc., New York, NY, USA.

[Weber, 2008] Weber, T. (2008). *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany.

[Werner, 1994] Werner, B. (1994). *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Universite Paris VII.

[Whittle and Cumming, 2000] Whittle, J. and Cumming, A. (2000). Evaluating environments for functional programming. *International Journal of Human-Computer Studies*, 52:847–878.

[Wilson et al., 2010a] Wilson, S., Fleuriot, J., and Smaill, A. (2010a). Automation for dependently typed functional programming. *To appear in: Special Issue of Fundamenta Informaticae on Dependently Typed Programming*.

[Wilson et al., 2010b] Wilson, S., Fleuriot, J., and Smaill, A. (2010b). Inductive proof automation for Coq. In *Proceedings of the 2nd Coq Workshop*, EPTCS.

[Xi, 1998] Xi, H. (1998). *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University.

[Xi, 1999a] Xi, H. (1999a). Dead code elimination through dependent types. *Lecture Notes in Computer Science*, 1551:228–242.

[Xi, 1999b] Xi, H. (1999b). Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris.

[Xi, 2000] Xi, H. (2000). Imperative programming with dependent types. In *LICS '00: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, page 375, Washington, DC, USA. IEEE Computer Society.

[Xi, 2001] Xi, H. (2001). Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, Boston.

[Xi, 2010] Xi, H. (2010). *The ATS Programming Language*.

[Xi and Harper, 2001] Xi, H. and Harper, R. (2001). A dependently typed assembly language. In *International Conference on Functional Programming*, pages 169–180.

[Xi and Pfenning, 1998] Xi, H. and Pfenning, F. (1998). Eliminating array bound checking through dependent types. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257.

[Xi and Pfenning, 1999] Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY.

Appendix A

Function and Type Definitions

The following sections give the definitions of the functions and types used in this thesis.

A.1 Peano Arithmetic

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.

Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (plus p m)
  end.

Infix "+" := plus.

Fixpoint minus (n m:nat) {struct n} : nat :=
  match n, m with
  | O, _ ⇒ 0
  | S k, O ⇒ S k
  | S k, S l ⇒ minus k l
  end.

Infix "−" := minus.
```

```
Fixpoint mult (n m:nat) : nat :=
  match n with
  | O ⇒ 0
  | S p ⇒ m + mult p  m
  end

Infix "∗" := mult.

Fixpoint pow (r n:nat) : nat :=
   match n with
   | O ⇒ 1
   | S n ⇒ r ∗ pow r n
   end.

Infix "ˆ" := pow.

Fixpoint max (n m:nat) : nat :=
  match n, m with
  | O, _ ⇒ m
  | S n', O ⇒ n
  | S n', S m' ⇒ S (max n' m')
  end.

Fixpoint min (n m:nat)  : nat :=
  match n, m with
  | O, _ ⇒ 0
  | S n', O ⇒ 0
  | S n', S m' ⇒ S (min n' m')
  end.

le_gt_dec : ∀ n m: nat), {n ≤ m} + {n > m}

nat_eq_dec : ∀ (n m:nat), {n = m} + {n ≠ m}
```

A.2 Lists

```
Inductive list (A:Type) : Type :=
  | nil  : list A
  | cons : A → list A → list A.

Inductive vect (A:Set) : nat → Set :=
  | vnil : vect A O
  | vcons : ∀ (n:nat), A → vect A n → vect A (S n).

Fixpoint length (A:Type) (a:list A) : nat :=
  match a with
  | [] ⇒ O
  | h :: t ⇒ S (length t)
  end.

Fixpoint app (A:Type) (a b:list) : list A :=
  match a with
  | [] ⇒ b
  | h :: t ⇒ h :: app t b
  end.

Fixpoint rev (a:list A) : list A :=
  match a with
  | nil ⇒ nil
  | h :: t ⇒ rev t ++ h :: nil
  end.

Fixpoint sum (a:list nat) : nat :=
  match a with
  | [] ⇒ 0
  | h::t ⇒ h + sum t
  end.

Fixpoint fold_left (A B : Type)
```

```
  (f : A → B → A) (a:list B) (i:A) : A :=
  match a with
  | nil ⇒ i
  | cons h t ⇒ fold_left f t (f i h)
  end.

Fixpoint list_count (a:list nat) (x:nat) : nat :=
  match a with
  | nil ⇒ 0
  | h::t ⇒
    if nat_eq_dec h x then
      S (list_count t x) else list_count t x
  end.

Notation list_perm x y :=
  (∀ n, list_count x n = list_count y n)
```

A.3 Binary Trees

```
Inductive btree (A:Type) : Type :=
  | empty : btree
  | node : A → btree → btree → btree.

Fixpoint inorder (a:btree A) : list A :=
  match a with
  | empty ⇒ []
  | node v l r ⇒ (inorder l) ++ [v] ++ (inorder r)
  end.

Fixpoint fold_left (A B:Type)
  (f:B→A→B) (l : btree A) (i : B) : B :=
  match l with
  | empty ⇒ i
  | node v l r ⇒ fold_left f r (f (fold_left f l i) v)
  end.
```

```
Fixpoint btree_count (a : btree nat) (x : nat) : nat :=
  match a with
  | empty ⇒ 0
  | node v l r ⇒
    let countlr := tree_count l x + tree_count r x in
    if nat_eq_dec v x then S countlr else countlr
  end.

Notation btree_perm x y :=
  (∀ n, btree_count x n = btree_count y n)
```

A.4 IsaPlanner Theorem Corpus Definitions

The following Coq definitions were used for the IsaPlanner theorem corpus experiment (see §7.14):

```
Fixpoint last (l:list A) (d:A) : A :=
  match l with
  | [] ⇒ d
  | [a] ⇒ a
  | a :: l ⇒ last l d
  end.

Fixpoint less_eq m n {struct m} : Prop :=
  match m, n with
  | 0, _ ⇒ True
  | S m', 0 ⇒ False
  | S m', S n' ⇒ (less_eq m' n')
  end.

Fixpoint less m n {struct m} : Prop :=
  match m, n with
  | _, 0 ⇒ False
  | 0, S n' ⇒ True
  | S m', S n' ⇒ (less m' n')
  end.

Lemma less_eq_dec :
  ∀ (x y : nat), {less_eq x y} + {~ less_eq x y}.
induction x; induction y;  simpl in *; try tauto.
apply IHx.
Defined.

Lemma less_dec :
  ∀ (x y : nat), {less x y} + {~ less x y}.
induction x; induction y;  simpl in *; try tauto.
apply IHx.
Defined.

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

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

Inductive btree (A:Type) : Type :=
  | empty : btree A
  | node : A → btree A → btree A → btree A.

Fixpoint mirror (A:Type) (a : btree A) : btree A :=
  match a with
  | empty ⇒ empty A
  | node v l r ⇒ node v (mirror r) (mirror l)
  end.

Fixpoint height (A:Type) (a : btree A) : nat :=
  match a with
  | empty ⇒ 0
  | node v l r ⇒ 1 + max (height l) (height r)
  end.

Fixpoint drop (A:Type) (n:nat) (a : list A)
  {struct a} : list A :=
  match a with
  |  nil ⇒ []
  | h::t ⇒
    match n with
    | O ⇒ a
    | S p ⇒ drop p t
    end
  end.

Fixpoint take (A:Type) (n:nat) (a : list A)
  {struct a} : list A :=
  match a with
  | nil ⇒ []
  | h::t ⇒
    match n with
    | O ⇒ []
    | S p ⇒ h::(take p t)
    end
  end.

Fixpoint takeWhile (A:Type) (P:A→bool)
  (a : list A) {struct a} : list A :=
  match a with
  |  nil ⇒ []
  | h::t ⇒
    if P h then h::(takeWhile P t) else []
  end.

Fixpoint dropWhile (A:Type) (P:A→bool)
  (a : list A) {struct a} : list A :=
  match a with
  | nil ⇒ []
  | h::t ⇒
    if P h then (dropWhile P t) else a
  end.

Fixpoint butlast (A:Type) (a : list A) : list A :=
  match a with
  | nil ⇒ []
  | h::t ⇒
    match t with
    | [] ⇒ []
    | _ ⇒ h::(butlast t)
    end
  end.

Fixpoint member (A:Type)
  (eqA : ∀ (x y : A), {x = y} + {x ≠ y})
  (x : A) (a : list A) {struct a} : Prop :=
  match a with
  | nil ⇒ False
  | h :: t ⇒
    if eqA x h then True else (member eqA x t)
  end.

Fixpoint insert (x:nat) (a:list nat) : list nat :=
  match a with
  | nil ⇒ [x]
  | h::t ⇒
```

```
    if less_dec x h then x::a else h::(insert x t)
  end.

Fixpoint insert_1' (x:nat) (a:list nat) : list nat :=
  match a with
  | nil ⇒ [x]
  | h::t ⇒
    if eq_nat_decide x h then x::t else h::(insert_1' x t)
  end.

Fixpoint sorted (a : list nat) : Prop :=
  match a with
  | nil ⇒ True
  | h1::t1 ⇒
    match t1 with
    | nil ⇒ True
    | h2::t2 ⇒
      if less_eq_dec h1 h2 then (sorted t1) else False
    end
  end.

Fixpoint count (A:Type)
  (eqA : ∀ (x y:A), {x = y} + {x ≠ y}) (x : A) (a : list A) :
    nat :=
  match a with
  | nil ⇒ 0
  | h :: t ⇒
    if eqA x h then S (count eqA x t) else (count eqA x t)
  end.

Fixpoint insort (x:nat) (a:list nat) {struct a} : list nat :=
  match a with
  | nil ⇒ [x]
  | h::t ⇒
    if less_eq_dec x h then x::a else h::(insort x t)
  end.

Fixpoint sort (a:list nat) : list nat :=
  match a with
  | nil ⇒ []
  | h::t ⇒ insort h (sort t)
  end.

Fixpoint zip (l : list A) (l' : list B) : list (A*B) :=
  match l,l' with
  | x::tl, y::tl' ⇒ (x,y)::(zip tl tl')
  | _, _ ⇒ nil
  end.
```

Appendix B

Case Study Results

The table in this appendix gives the results of running our prover against the proof obligations generated from the case study programs from Chapter 9. The labels in the header of the table denote which configuration of our prover was used for each set of results:

DiscoverAndCache (DAC): The lemmas found by lemma discovery tool are available for use by the prover. The lemma cache is cleared after each verification task (where the tasks are attempted in the order given in the table). This is the configuration used when conducting the case studies described in detail in Chapter 9.

DiscoverOnly (DO): The lemmas found by lemma discovery tool are available for use by the prover. Lemmas are *not* cached after proof attempts.

CacheOnly (CO): The lemmas found by lemma discovery tool are *not* available for use by the prover. Lemmas are cached after proof attempts. The lemma cache is *not* cleared after each verification task (where the tasks are attempted in the order given in the table).

BasicDefs (BD): The prover only uses basic definitions during proofs (i.e. it is not allowed to use cached lemmas).

The ST label denotes the results of our prover when ran against simply typed versions of the goals from the case studies (see §9.8.1) using the same configuration as BD above. The IsaP label denotes results from IsaPlanner (configured to only use basic definitions) running against Isabelle versions of these goals.

Goal	DAC	DO	CO	BD	ST	IsaP
Tail recursive sum (without fold):						
1	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	×
Tail recursive sum without fold (variant):						
3	✓	✓	✓	✓	✓	×
4	✓	✓	✓	✓	✓	×
Tail recursive sum with fold:						
5	✓	✓	✓	✓	✓	×
Tail recursive factorial without fold:						
6	✓	✓	✓	✓	✓	✓
7	✓	✓	✓	✓	✓	×
8	✓	✓	✓	✓	✓	×
Tail recursive factorial without fold (variant):						
9	✓	✓	✓	✓	✓	✓
10	✓	✓	✓	×	×	×
11	✓	✓	✓	✓	✓	×
Tail recursive factorial with fold:						
12	×	×	×	×	×	×
Tail recursive factorial with fold (variant):						
13	×	×	×	×	×	×
14	✓	✓	✓	✓	✓	×
Tail recursive inorder without fold:						
15	✓	✓	✓	✓	✓	×
16	✓	✓	✓	✓	✓	×
Tail recursive inorder with fold:						
17	×	×	×	×	×	×
Tail recursive inorder with fold (variant):						
18	✓	✓	✓	✓	✓	×
19	✓	✓	✓	✓	✓	×
Insertion sort (length property):						
20	✓	✓	✓	✓	✓	✓
21	✓	✓	✓	✓	✓	×
Insertion sort (length property variant):						
22	✓	✓	✓	✓	✓	×

Continued on next page

Goal	DAC	DO	CO	BD	ST	IsaP
Insertion sort (permutation property):						
23	✓	✓	✓	✓	✓	×
24	✓	✓	✓	✓	✓	×
Insertion sort (permutation property variant):						
25	✓	✓	✓	✓	✓	×
Treesort (length property):						
26	✓	✓	✓	✓	✓	✓
27	✓	✓	✓	✓	✓	✓
28	✓	✓	✓	✓	✓	×
29	✓	✓	✓	✓	✓	×
Treesort (permutation property):						
30	✓	✓	✓	✓	✓	×
31	✓	✓	✓	✓	✓	×
32	✓	✓	✓	✓	✓	×
33	×	×	×	×	×	×
Treesort (permutation property variant):						
34	✓	✓	✓	✓	✓	×
35	✓	✓	✓	✓	✓	×
36	×	×	×	×	×	×
37	✓	✓	✓	✓	✓	×
Quicksort (length property):						
38	✓	✓	✓	✓	✓	✓
39	✓	✓	✓	✓	✓	×
40	×	×	×	×	×	×
41	×	×	×	×	×	×
42	✓	✓	✓	✓	✓	×
Quicksort (permutation property):						
43	✓	✓	✓	✓	✓	×
44	✓	✓	✓	✓	✓	×
45	×	×	×	×	×	×
46	×	×	×	×	×	×
47	×	×	×	×	×	×
Binary adder using inductive families:						
48	✓	✓	✓	×	×	×
49	✓	✓	✓	×	×	×

Continued on next page

Goal	**DAC**	**DO**	**CO**	**BD**	**ST**	**IsaP**
50	✓	✓	✓	✓	✓	✓
51	✓	✓	✓	✓	✓	×
52	✓	✓	✓	×	×	×
53	✓	✓	✓	✓	✓	×
54	✓	✓	✓	✓	✓	×
Binary adder using inductive families (variant):						
55	✓	✓	✓	×	×	×
56	✓	✓	✓	×	×	×
57	✓	✓	✓	✓	✓	×
58	✓	✓	✓	×	×	×
59	✓	✓	✓	✓	✓	×
60	✓	✓	✓	✓	✓	×
Binary adder using subset types:						
61	✓	✓	✓	×	×	×
62	✓	✓	✓	×	×	×
63	✓	✓	✓	✓	×	×
64	✓	✓	✓	✓	✓	×
65	✓	✓	✓	✓	✓	×
66	✓	✓	✓	✓	✓	×
67	×	×	×	×	×	×
Total successes	56	56	56	47	46	8
Total failures	11	11	11	20	21	59
Time (s)	107.51	103.83	138.71	122.93	149.59	341.25

Appendix C

IsaPlanner Theorem Corpus Experiment

The following table contains the experimental results generated from running our prover against a theorem corpus that has used to evaluate IsaPlanner (see §7.14). For successfully automated theorems (indicated with a tick mark), the time indicates how long our prover took to find a proof. For theorems that could not be automated (indicated with a cross mark), the time indicates how long the prover took to fail.

No.	Theorem	Result	Time (s)
01	m − m = 0	✓	0.01
02	n − (n + m) = 0	✓	0.06
03	n + m − n = m	✓	0.06
04	k + m − (k + n) = m − n	✓	0.08
05	i − j − k = i − (j + k)	✓	0.07
06	less_eq n 0 ↔ n = 0	×	0.47
07	less_eq n (n + m)	✓	0.20
08	less i (S (i + m))	✓	0.01
09	max a b = max b a	✓	0.06
10	max (max a b)c = max a (max b c)	✓	0.09
11	max a b = a ↔ less_eq b a	×	0.67
12	max a b = b ↔ less_eq a b	×	0.12
13	min a b = min b a	✓	0.06
14	min (min a b) c = min a (min b c)	✓	0.07
15	min a b = a ↔ less_eq a b	×	0.80

Continued on next page

No.	Theorem	Result	Time (s)
16	min a b = b ↔ less_eq b a	×	0.83
17	drop 0 xs = xs	✓	0.02
18	drop (S n) (x :: xs) = drop n xs	✓	0.00
19	drop n (map f xs) = map f (drop n xs)	✓	11.77
20	len (drop n xs) = len xs − n	✓	0.09
21	take 0 xs = []	✓	0.01
22	take (S n) (x :: xs) = x :: take n xs	✓	0.00
23	take n (map f xs) = map f (take n xs)	✓	14.26
24	take n xs ++ drop n xs = xs	✓	0.22
25	zip [] ys = []	✓	0.00
26	zip (x :: xs) ys = **match** ys **with** [] ⇒ [] | (z :: zs) ⇒ (x, z) :: zip xs zs **end**	✓	0.00
27	zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys	✓	0.00
28	height (mirror t) = height t	✓	0.20
29	member x (l ++ (x :: []))	✓	0.06
30	∼member x (delete x l)	✓	0.08
31	member x l →member x (l ++ t)	×	0.01
32	member x t →member x (l ++ t)	×	0.01
33	member x (insert x l)	✓	0.45
34	member x (insert_1' x l)	✓	0.25
35	len (insert x l) = S (len l)	✓	0.21
36	len (sort l) = len l	✓	0.32
37	xs = [] → last (x :: xs) default = x	✓	0.00
38	1 + count n l = count n (n :: l)	✓	0.01
39	n = x → 1 + count n l = count n (x :: l)	✓	0.01
40	count n l + count n m = count n (l ++ m)	✓	0.22
41	count n (x ++ (n :: [])) = S (count n x)	✓	0.19
42	count n (h :: []) + count n t = count n (h :: t)	✓	0.00
43	less_eq (count n l) (count n (l ++ m))	×	0.26
44	dropWhile (fun _ ⇒ false) xs = xs	✓	0.02
45	takeWhile (fun _ ⇒ true) xs = xs	✓	0.04
46	takeWhile P xs ++ dropWhile P xs = xs	✓	1.81
47	filter P (xs ++ ys) = filter P xs ++ filter P ys	✓	0.21
48	m + n − n = m	×	0.38
49	S m − n − S k = m − n − k	×	10.30

Continued on next page

No.	Theorem	Result	Time (s)
50	less i (S (m + i))	×	0.06
51	less_eq n (m + n)	×	0.06
52	less_eq m n $\rightarrow$ less_eq m (S n)	×	0.02
53	drop n (drop m xs) = drop (n + m) xs	×	0.14
54	drop n (xs ++ ys) = drop n xs ++ drop (n − len xs) ys	×	0.29
55	drop n (take m xs) = take (m − n)(drop n xs)	×	0.14
56	drop n (zip xs ys) = zip (drop n xs) (drop n ys)	×	0.19
57	rev (drop i xs) = take (len xs − i) (rev xs)	×	0.15
58	rev (take i xs) = drop (len xs − i) (rev xs)	×	0.12
59	rev (filter P xs) = filter P (rev xs)	×	0.26
60	take n (xs ++ ys) = take n xs ++ take (n − len xs) ys	×	0.23
61	take n (drop m xs) = drop m (take (n + m) xs)	×	0.16
62	take n (zip xs ys) = zip (take n xs) (take n ys)	×	0.16
63	less_eq (len (filter P xs)) (len xs)	×	1.41
64	zip (xs ++ ys) zs = zip xs (take (len xs) zs) ++ zip ys (drop (len xs) zs)	×	0.53
65	zip xs (ys ++ zs) = zip (take (len ys) xs) ys ++ zip (drop (len ys) xs) zs	×	0.43
66	len xs = len ys $\rightarrow$ zip (rev xs) (rev ys) = rev (zip xs ys)	×	0.13
67	less_eq (len (delete x l)) (len l)	×	0.14
68	less x y $\rightarrow$ member x (insert y l) = member x l	×	12.59
69	x $\neq$ y $\rightarrow$ member x (insert y l) = member x l	✓	2.50
70	sorted l $\rightarrow$ sorted (insert x l)	×	0.01
71	sorted (sort l)	×	0.03
72	last (xs ++ (x :: [])) default = x	×	0.07
73	xs $\neq$ [] $\rightarrow$ last (x :: xs) default = last xs default	×	0.02
74	ys0 = [] $\rightarrow$ last (xs ++ ys0) default = last xs default	✓	0.04
75	ys $\neq$ [] $\rightarrow$ last (xs ++ ys) default = last ys default	×	0.25
76	last (xs ++ ys) default = **match** ys **with** [] $\Rightarrow$ (last xs default) | _ $\Rightarrow$ (last ys default) **end**	✓	0.13
77	less n (len xs) $\rightarrow$ last (drop n xs) default = last xs default	×	0.25
78	butlast (xs ++ (x :: [])) = xs	×	0.08
79	xs $\neq$ [] $\rightarrow$ butlast xs ++ (last xs default :: []) = xs	×	0.05
80	butlast (xs ++ ys) = **match** ys **with** [] $\Rightarrow$ (butlast xs) | _ $\Rightarrow$ (xs ++ butlast ys) **end**	✓	0.15

Continued on next page

No.	**Theorem**	**Result**	**Time (s)**
81	butlast xs = take (len xs − 1) xs	×	0.09
82	len (butlast xs) = len xs − 1	×	0.07
83	less_eq (len (delete x l)) (len l)	×	0.10
84	count n t + count n (h :: []) = count n (h :: t)	✓	0.14
85	count n l = count n (rev l)	×	0.67
86	count x l = count x (sort l)	×	1.65
87	n $\neq$ h $\rightarrow$ count n (x ++ (h :: [])) = count n x	✓	0.29