Computer-certified Proofs

Module: MATH5004M
Student ID: 200684195
Supervisor: Dr Nicola Gambino

Ismail Suleman

July 7, 2017

Contents

Chapter 1

Introduction

Coq is a formal proof assistant which provides an environment in which one can:

1. Write programs that can then be converted into code in another programming language eg Ocaml
2. Write mathematical theorems and proofs of those theorems (with the Coq interpreter verifying the proofs)

Both of these aspects can be combined to create certified programs, where properties of the program can be formally verified.

In this project, I will explain how to use Coq and then rewrite and restructure some theorems and their proofs from the standard library relating to natural numbers. Due to the size and complexity of the standard library, only a subset will be rewritten. Also, many advanced aspects of Coq will not be studied, and theorems/proofs using them will not be considered as part of this project.

1.1 Reading this report

This report is intended to be read from front-to-back. However, Coq is a complex program, and often it may be necessary to revisit previous parts of the report to get a full understanding.

This report also includes many pieces of example code. These are pretty-printed in this report for readability. However, bear in mind that this results in some abbreviations in the code shown here compared to what one would enter into Coq. These are detailed in this table:

Abbreviation	What would be entered into Coq
$\vee$, $\wedge$	\/ , /\

Abbreviation	What would be entered into Coq
$\leftarrow$, $\rightarrow$	<- , ->
$\Rightarrow$	=>
$\neg$	~ (tilde)

Throughout this report, worked exercises are included from [1]. Where this is done, the exercise number will be given (some of the exercises are modified slightly).

Chapter 2

Overview of the Coq system

The Coq system is described in detail in [2]. In this section, which is based on [1,2] we focus on giving a more readable explanation of only the concepts relevant to this report.

2.1 Basic concepts

2.1.1 Language

Coq is designed principally as an interactive language. This means that commands are passed in sequence to a running `coqtop` program, which prints the output of the commands as they are executed. This is in contrast to other programming languages (eg Java, Haskell) which are designed to be compiled into an executable.

The language that Coq is given commands in is called *Gallina*. Note that each command starts with a capital letter and ends with a full stop (eg `Check` 5.), in a similar fashion to written languages, such as English.

2.1.2 Basic Type Theory

This section covers some fundamental type theory of Coq. More advanced concepts are introduced in section 2.2.

2.1.2.1 Terms and types

In set theory, we have the concept of a set as being a collection of objects with no duplicates allowed. Objects can be elements of a set, and sets can be elements of other sets, so forming a hierarchy.

In an analogous manner, in Coq we have *terms* which have a *type* (the concept of a type in Coq is similar to that in most other programming languages eg Java, Python, Haskell). A type is analogous to a set. Types themselves are terms, and types themselves have types, which is analogous to sets being elements of other sets. Note that in Coq circular references are not allowed. That is, if A is of type B, then B cannot be of type A.

Please note that in this report, the words *terms*, *elements* and *objects* are used interchangeably.

2.1.2.2 Type hierarchy

In Coq the hierarchy of terms and their types is an infinite hierarchy, and is shown in figure 2.1. This hierarchy fulfils the following conditions:

1. every term has a type
2. we do not have any circular references in the hierarchy

There are 2 main sections of this hierarchy:

- Elements of the type `Set`. This is where terms and types related to programs reside.
- Elements of the type `Prop`. This is where terms and types related to propositions and proofs reside.

Both `Set` and `Prop` are of type `Type(0)`. `Type(0)` is of type `Type(1)` and in general, `Type(i)` is of type `Type(j)` where `i < j`. Also note that the Coq interpreter by default abbreviates `Type(i)` to just `Type`, which means that the Coq interpreter will say that `Type` is of type `Type`.

`Set`, `Prop` and `Type(i)` (forall `i`) are known as *sorts* (ie a type of a type is a sort). `Type(i)` (forall `i`) are also known as *universes*.

2.1.2.3 Function types

If `A` and `B` are types, then we can construct a new type `A → B`. This is called a *function type*. In this case, this is the type of a function that takes as input a term of type `A` and returns a term of type `B`.

Functions can only return one object, but that object could be made up of multiple objects (see section 2.2.8 for more details). Functions can, however, accept multiple arguments. For instance, `A → B → A` is the type of a function

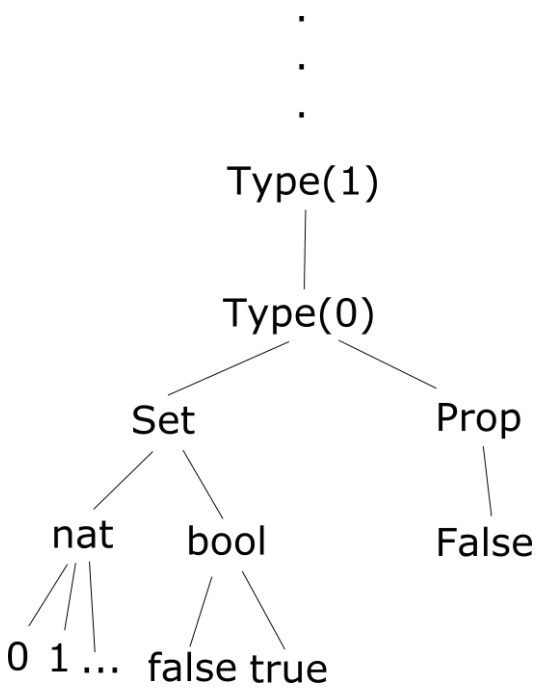

Figure 2.1: Hierarchy of types in Coq

that takes as input both an element of type `A` and an element of type `B` and returns an element of type `A`. Examples of functions with such a type are:

- a function that returns the first argument it is passed and ignores the second one
- a function that returns a particular term of type `A` regardless of the arguments it is passed
- a function that does some computation involving the 2 input arguments in order that results in an element of type `A`

Coq also has the concept of *currying*, which is where a function can be called with fewer than the required arguments, and this produces a new function that accepts the remaining arguments and exhibits the same behaviour as the original function. For example, if we start with a function of type `A → B → A` and call it with an element of type `A`, then we get a function of type `B → A`. Calling this function with an element of type `B` then gives us an element of type `A`.

2.1.2.4 Higher-order functions

Functions themselves are terms in Coq, and can be passed as inputs to other functions or returned by other functions (this concept is known as *higher-order*

functions). To illustrate this, two examples are presented:

- the type (`A` → `A`) → `B` is the type of a function that takes as input a function, and returns a term of type `B`. The function that is given as input must have type `A` → `A`.
- the type `A` → (`A` → `B`) is the type of a function that takes as input an element of type `A` and returns a function of type `A` → `B`. This function that is returned can then be called with an element of type `A` and will then return an element of type `B`.

The type presented in the second example, `A` → (`A` → `B`), can be also written as `A` → `A` → `B` (ie → is right-associative). This illustrates an important concept that instead of thinking of functions in Coq as accepting potentially multiple arguments, one can instead think of functions as always taking one input and having one output. That output could be a function that takes an argument and outputs either a value or another function.

2.1.3 Basic Gallina Commands

As mentioned in section 2.1.1, Coq is used interactively by entering commands into a running Coq interpreter, which then carries out the commands. In this section, some basic Gallina commands are discussed (some more advanced commands are discussed in section 2.2). Note that many aspects of Coq are interlinked such that understanding a single part can require understanding how it relates to other parts of Coq. Thus it may be necessary to revisit previous parts of this section.

2.1.3.1 Comments

Comments are ignored by Coq and are used to add explanatory information to source code.

Comments in Coq source code are written between `(*` and $*)$. For example:

```
(*This is a comment*)
```

is an example of a comment. Comments can also be nested (unlike some other programming languages, such as Javascript), for example:

```
1 (* This is a comment (*And this is a comment in a comment*)*)
```

When nesting comments, one must make sure that all opening tags `(*` are closed by a matching $*)$.

2.1.3.2 `Check`

`Check` can be passed a term and it prints the term and the type of the term. For example:

```
Check 5.
```

prints

```
5
     : nat
```

There also exists a related command called `Print`. This is explained in section 2.1.3.4

2.1.3.3 Scopes

The notation used to construct terms can be customised in Coq. This can be done using the `Notation` command, which can be used as follows:

```
Notation "x = y" := (eq x y).
```

This essentially tells Coq that whenever it sees code that looks like `x = y`, it should treat it as if it was written `eq x y`.

These notations can be grouped into *scopes*. For example:

```
Notation "x = y" := (eq x y): nat_scope.
```

puts our previous notation into the `nat_scope` (scopes are created implicitly on their first usage).

In the standard library there are a number of scopes that have been defined, such as `nat_scope`, `Z_scope`, `type_scope`.

Scopes can be either open or closed. Additionally, open scopes are ordered based on how recently they were opened. When Coq interprets notation when evaluating a term, it looks in the most recent scope that is open first to see if the notation has been defined there, and then looks in scopes that were opened successively earlier.

By default, on starting the Coq interpreter the scopes `core_scope`, `type_scope` and `nat_scope` are opened, in that order.

To see what is inside a particular scope, use the `Print Scope` command. For example:

```
Print Scope nat_scope.
```

prints

```
Scope nat_scope
Delimiting key is nat
Bound to class nat
"x >= y" := ge x y
"x > y" := gt x y
"x <= y <= z" := and (le x y) (le y z)
"x <= y < z" := and (le x y) (lt y z)
"n <= m" := le n m
"x < y <= z" := and (lt x y) (le y z)
"x < y < z" := and (lt x y) (lt y z)
"x < y" := lt x y
"x - y" := Nat.sub x y
"x + y" := Nat.add x y
"x * y" := Nat.mul x y
```

Analysing this output, we see the following:

- The first line just gives the name of the scope
- The second line tells us what the *delimiting keys* are for this scope. These are shorter versions of the name that can be used to refer to the scope. We will see an example of using them later in this section.
- We will ignore the third line as it is unnecessary for our discussion
- The remaining lines each show how Coq will rewrite various syntaxes it sees. For example, as the `nat_scope` is open by default, the following code:

```
Check (5 + 5).
```

prints

```
10: nat
```

as $5 + 5$ is rewritten as `Nat.add` 5 5 which is then evaluated to give 10.

The usage of a particular scope can be done by using the syntax `term%delimiting_key`. For example:

```
Require Import ZArith.
Check 5%Z.
```

prints

```
5%Z
     : Z
```

where `Z` is the delimiting key for the scope `Z_scope`.

This example uses imports, which are briefly explained in section 2.1.3.6

2.1.3.4 Declaring and defining variables

In section 2.1.2, terms and types were discussed. Like many other programming languages, in Coq we can create variables that have a type and optionally are assigned a term. There are four related commands that can be used to create variables:

- `Parameter`
- `Definition`
- `Variable`
- `Let`

`Parameter` declares a variable, which must have a type. For example:

```
Parameter example: nat.
```

declares a variable called example, that is a natural number (ie has type `nat`). Note that we have not actually associated any particular natural number to the variable example, so the variable represents a general natural number rather than any specific one.

If we wish to create a variable for a specific term, then we can use `Definition`. For example:

```
Definition example2: nat := 5.
```

defines a variable called example2, which is a natural number (so has type `nat`), and more specifically is the natural number 5.

`Variable` behaves in a very similar way to `Parameter`, and `Let` behaves in a similar way to `Definition`. The difference between `Parameter`/`Definition` and `Variable`/`Let` is that the former create variables in the *global context* whereas the latter create variables in the current *section*. This will be explained in more detail in section 2.1.3.5.

Most of these commands have synonyms as detailed in the following table:

Command	Synonyms
`Parameter`	`Axiom`, `Conjecture`
`Definition`	`Example`
`Variable`	`Hypothesis`
`Let`	None

Note that many of these commands have plural versions of the command available eg `Parameters`, `Variables`, `Hypotheses`.

These synonyms can be used to make Coq code more readable and make the intention of the code clearer.

Also note that a variable can only be created once. Once created, a variable is immutable and cannot be reassigned. For example, the code:

```
Definition will_fail : nat := 5.
Definition will_fail : nat := 4.
```

raises the following error:

```
Error: will_fail already exists.
```

There exists a command called `Print`, which when passed a variable, prints information about the variable. For example:

```
Definition test := 5.
Print test.
```

prints

```
test = 5
     : nat
```

2.1.3.5 Sections

Sections are a feature of Coq that helps structure Coq source code. Two reasons for using sections are that:

- they allow one to group related source code together under a name
- they allow for simplifying the definitions of multiple related functions.

Sections are started using the `Section` command and are ended using the `End` command. When a section is ended, any variables created in the current section (ie using the commands `Variable`, `Let` or their synonyms) are discharged. Sections can also be nested. All of this is best illustrated by an example:

```
Section first.
  Variable X: nat.
  Let two: nat := 2.
  Definition x_plus_two: nat := X + two.
  Section second.
    Variable Y: nat.
    Definition x_times_y: nat := X * Y.
  End second.

  Print x_times_y.

  (* This prints:
x_times_y = fun Y : nat ⇒ X * Y
     : nat → nat

Argument scope is [nat_scope]

Notice how the variable Y has been discharged (as Y was defined only for
    the section it was created in), but X hasn't (as the section it was
    defined in is still open). Notice also how the process of
    discharging has turned x_times_y into a function that accepts one
    parameter, Y. *)

End first.

Print x_plus_two.
(* This prints:
x_plus_two = fun X : nat ⇒ let two := 2 in X + two
     : nat → nat

Argument scope is [nat_scope]

Here 'X' and 'two' have been discharged. This has turned x_plus_two into
    a function taking 'X' as a parameter. As 'two' was defined in the
    section only, its definition has been moved into the body of the
    function. *)

```

```
Print x_times_y.
(* This prints:
x_times_y = fun X Y : nat ⇒ X * Y
     : nat → nat → nat

Argument scopes are [nat_scope nat_scope]

Both X and Y have been discharged. *)
```

Note that sections are not analogous to modules, which are a completely separate concept in Coq, and are covered in section 2.1.3.6.

2.1.3.6 Imports

Whilst modules are not used in the partial rewrite of the standard library that is covered in section 3, they are used in the standard library itself, so it is necessary to at least cover the basic concepts in order to study the standard library, although we won't go into detail.

Like many programming languages, Coq has a module mechanism which allows separating code over files. Modules are started using the `Module` command and ended using the `End` command. Like sections, modules can also be nested inside each other and there can be multiple modules defined in one file (this is unlike many programming languages which consider a file or directory as a module). Modules can also be imported using the `Require Import` command (note there are several similar commands for importing modules which won't be covered here).

Coq also has the concept of a *module type*. A module type has a name and a set of variables that have types. Other modules can then implement a module type, which means that they must have all the corresponding variables with the correct types defined (they can also define additional variables). A single module can implement multiple module types.

Both modules and module types can be parametrised, which means that one can essentially define a function that takes as input arguments and returns a module or module type. These are known as *functors* (not to be confused with the concept of functors in some other functional programming languages, where the word functor describes generalised mapping over certain types). Functors are used in various places in the standard library to define several modules that have similar functions and theorems.

2.1.4 Summary of basic Gallina commands

This section is a summary of the basic Gallina commands described in the last section.

Check Display the supplied term on screen, along with its type.

```
Check 5.
```

Print Display information about the supplied variable.

```
Print nat.
```

Parameter Create a global declaration of a variable. Only the type of the variable is specified. No value can be assigned to the variable within or after the command.

```
Parameter Q: Prop.
```

Definition Create a global definition of a variable. This is the same as the **Parameter** command but a value is assigned to the variable. A type can be specified explicitly or inferred.

```
Definition x := false.

Definition y: bool := false
```

Variable Create a local declaration of a variable. Essentially, this behaves the same as **Parameter** but the variable is only available inside the current section.

```
Variable A: Prop.
```

Let Create a local definition of a variable. Essentially, this behaves the same as **Definition** but the variable is only available inside the current section.

```
Let x: nat := 5.
```

Section / End Start / end a section. Note that sections can be nested. Within sections, one can create and use local and global variables. Outside a section, any global variables that used locally declared variables become functions that take as input values that take the place of the locally declared variables (as shown in the following example).

```
Section test.
  Variable X: nat.
  Definition x_plus_two: nat := X + 2.
End test.

Print x_plus_two.
(* This prints x_plus_two = fun X : nat ⇒  X + 2
     : nat →  nat *)
```

Require Import Import a module.

```
Require Import Arith.
```

Theorem / Proof Declare an element of a given type, with a proof block following that constructs an element of that type. Generally used when writing theorems in Coq, with a proof of the theorem.

See section 2.2.10 for further explanation.

2.1.5 Functions

Functions can be created by using the `fun` keyword. For example:

```
Definition add_one: nat → nat := fun (x: nat) ⇒ x + 1.
```

defines a variable `add_one`, which has a type `nat → nat` (ie it is a function that takes as input a natural number and returns another natural number). After the `fun` keyword, we specify that this function takes as input one argument that must be a natural number. The body of the function comes after ⇒ . In this case, we simply add 1 to the input number.

This can be rewritten in a variety of equivalent ways:

- `Definition add_one := fun x ⇒ x + 1.`
  - Here we use Coq's type inference which allows us to avoid having to specify types explicitly in many places. In such situations, the Coq interpreter will look at the terms that it does know the types of, and see how they are used to infer the types of other variables that have not got explicitly specified types. This allows for shorter code, but can if used excessively make code harder to read.
- `Definition add_one x: nat := x + 1.`
  - Instead of using the `fun` keyword, we can use the shorthand shown in this example.
- `Definition add_one x := x + 1.`
  - This example is the same as the last example, but where we have chosen to not explicitly specify the type of the input argument, again taking advantage of Coq's type inference.

As mentioned in section 2.1.2.3, functions can also take multiple parameters. For example:

```
Definition add_one_optionally: nat → bool → nat := fun (x: nat) (add_one:
    bool) ⇒ if add_one then x + 1 else x.
```

defines a function that takes two arguments as input and returns a natural number. The second argument is a boolean (ie either `true` or `false`), and if that value equals `true`, then we return `x + 1`, otherwise we return the input number `x` unchanged. Note that this example showcases some new syntax, an if-else statement. This behaves in a very similar way to if-else statements in other functional programming languages (in particular Haskell).

Again, the definition of `add_one_optionally` can be rewritten in a variety of equivalent ways:

- `Definition add_one_optionally := fun x (add_one: bool) ⇒ if add_one then x + 1 else x.`
  - Here the type of `add_one_optionally` has been omitted, along with the type of `x`. The type of `add_one` does not get inferred by Coq and needs to be explicitly specified.
- `Definition add_one_optionally x (add_one: bool) := if add_one then x + 1 else x.`
  - Here we have used the shorthand syntax instead of using the `fun` keyword.

2.1.6 A brief study of example code

Now that some basic aspects of Coq have been covered, some example code will be given here using what has been described in this section so far along with some new concepts that will be covered in the next section (section 2.2).

```
(* part of exercise 4.3, p93 *)
Section A_declared.
  Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).

  Theorem all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
  Proof (fun H a b ⇒ H b a).

  Theorem all_perm': (forall a b: A, R a b) → forall a b: A, R b a.
  Proof.
    intros H a b.
    apply H.
  Qed.
End A_declared.
```

The first line of this example,

```
(* Exercise 4.3, p93 *)
```

is a comment.

```
1 Section A_declared.
```

Here we start a new section, which is later closed by the command `End A_declared.`.

```
Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).
```

Here 4 variables, A, P, Q, and R are declared without an associated value. Variable A has type `Set`, P and Q have type `A → Prop` (ie a function that takes an element of type `A` and returns an element of type `Prop`) and R has type `A → A → Prop` (ie a function that takes 2 elements of type `A` and returns an element of type `Prop`).

```
Theorem all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
```

A *theorem* is an element of type `Prop`, also known as a proposition. Proving a theorem is then a matter of finding an element whose type is the theorem itself.

Here we are defining a variable called `all_perm`, which has a type `(forall a b: A, R a b) → forall a b: A, R b a`, which itself is of type `Prop`. This theorem states that if a binary predicate holds for all possible inputs `a` and `b` (where both inputs are of the same type), then the order in which the inputs `a` and `b` are supplied to the predicate doesn't affect whether or not the predicate holds, as it holds for all inputs (of the correct type).

```
Proof (fun H a b ⇒ H b a).
```

Here we give a proof of the preceding theorem, or in other words, we construct an element whose type is that of the preceding theorem. The element constructed here is a function which takes as input:

- `H` - a proof of the theorem `forall a b: A, R a b`, or equivalently, a function that takes as input 2 elements of the type `A` and returns an element of the type `R a b`
- `a` - an element of the type `A`
- `b` - an element of the type `B`

and then, considering `H` as a function, invokes `H` on `b` and `a` (in that order).

Proofs of theorems can be constructed in 2 ways:

1. By directly constructing the element that is the proof

2. By using *tactics* which provide a more user-friendly approach to constructing the element that constitutes the proof (see section 2.2.10 for more on tactics)

`fun H a b ⇒ H b a` is an example of directly constructing the proof. Later on in the example, the theorem `all_perm'` is defined. This theorem is the same theorem as `all_perm`, but the proof of the theorem has been constructed using tactics. By comparison with the direct proof of `all_perm`, it can be seen how the tactics correspond to constructing different aspects of the proof term:

- `intros H a b.` corresponds to specifying that we have a function that takes 3 arguments ie `fun H a b ⇒ body_of_function_still_to_be_determined`
- `apply H.` specifies that the body of this function consists of simply applying the function H.

2.2 More advanced concepts

2.2.1 Set

The `Set` type, as discussed briefly in section 2.1.2, has elements related to datatypes. Examples of such types of type `Set` are:

- `bool` which has only 2 elements of this type, `true` and `false`
- `nat` which contains all natural numbers
- Function types of the form `A → B`, where `A` and `B` are of type `Set`. These were discussed in section 2.1.2.3.

2.2.2 Prop

Also discussed briefly in section 2.1.2, the `Prop` type has as its elements types which can be thought of as propositions. Examples of such types are shown in the following code:

```
Variables (A B: Prop).
Definition example_proof (a: A): A := a.
```

- `A` and `B` are called *propositions*. Propositions are elements that are of type `Prop`.
- `example_proof` is a *proof*. A proof is an element whose type is a proposition. In this case, the type of `example_proof` is `A → A`, where `A` is of type `Prop`. Just as function types work within the `Set` type, the type `A → A` has type `Prop`. Note that in several places in Coq, the same Gallina syntax is used for constructing elements of the type `Set` and of the type `Prop`. Coq could

have also been created using different syntax for `Set` and `Prop`, but using the same syntax emphasises the relationship between programs and proofs (known as the Curry-Howard correspondence).

2.2.3 Negation

Negation in Coq is represented by a function `not` which is often invoked using the shorthand ¬. This function is of type `Prop → Prop`, and returns a proposition representing the negation of the input proposition. For example:

```
Variable A: Prop.
Definition example_usage_of_not: Prop := ¬A.
```

2.2.4 Inhabited types

In maths, there are many propositions that can be written down which are syntactically valid, but are not true (ie there exists no proof of them). In Coq, we can also write propositions which are not true eg `A → ¬A`. Syntactically, such propositions are valid, but there exist no proofs of them. In Coq, this means that there exist no elements which have that proposition as their type. Thus the notion of whether or not a proposition holds or not in Coq is equivalent to the notion of whether or not the type is inhabited (ie there exists an element with that type) or not.

A simple example of this is the term `False`, which is a proposition. There are no elements of type `False`, which is another way of saying that `False` is not provable (unless we have a contradiction). This is in fact used by the function `not`, which represents negation of a proposition, and is defined as:

```
Definition not: Prop → Prop := fun A : Prop ⇒ A → False.
```

Thus, proving the negation of a proposition is done by showing that if we have a proof of the proposition, then we have a contradiction, and so can construct a term of type `False`.

2.2.5 Predicates and Connectives

Also directly related to the type `Prop` are *predicates* and *connectives*.

- Predicates are functions that take any number of inputs whose type is of type `Set` and return a proposition. An example of a predicate is: `Variable H: nat → Prop`. The function `H` takes as input a number and returns a proposition. An example of such a function is a function is_prime, which would return a proposition asserting that the input number is a prime number.
- Connectives are functions that take any number of inputs which are propositions and return a proposition. An example of a connective is $\wedge$ or `and`, which comes built-in to Coq. `and` is a function of type `Prop → Prop → Prop` and returns a proposition which represents the logical conjunction of the two input propositions. Disjunction is reprsented by $\vee$ or `or`, which is a function of type `Prop → Prop → Prop`.

2.2.6 The Dependent Product

The dependent product is one of the main features that gives Coq a lot of expressiveness. The essential idea behind the dependent product is that the type of a function can depend on the input values. To construct such types, we need to use the `forall` syntax, as shown in this example:

```
Variables (A: Set) (B: A → Type).

Variable C: forall a: A, B a.
```

In this example, we have the following terms:

- `A` is an element of type `Set` (so `A` could be eg `nat` or `bool`)
- `B` is a function mapping elements of `A` to a type. `B` can also be thought of as a family of types indexed by elements of type `A`
- `C` is a function that takes one argument a. `C` is an example of the dependent product. The type of `C` depends on the value that it is passed. Note that `B` is an arbitrary function that returns a type.

Dependent products are used throughout Coq, for writing program specifications, propositions and proofs.

In section 2.1.6, the example code discussed included an example of the dependent product:

```
Variables (A: Set) (P Q: A → Prop) (R: A → A → Prop).
Hypothesis all_perm: (forall a b: A, R a b) → forall a b: A, R b a.
```

In this example, the function `R` is a binary predicate. `R` is then used in two dependent products in the type of `all_perm`. Note that `all_perm` itself can be

seen as simply a function that takes as input a function that has a dependent type and returns a function that has a dependent type.

The dependent product can be broken down into several different cases, as shown in the following table taken from [1] p. 92, describing terms constructed using the following rule:

$$\frac{\texttt{A: s} \qquad \texttt{a: A} \vdash \texttt{B: s'}}{\texttt{forall a: A, B: s''}}$$

where s, s' and s'' are given in this table:

Triplet (s, s', s'')	constraints	role
(s, s', s')	s, s' $\in$ {Set, Prop}	simple types
(Type(i), Prop, Prop)		Impredicativity in Prop
(s, Type(i), Type(i))	s $\in$ {Set, Prop}	dependence
(Type(i), Type(j), Type(k))	$(i \leq k, j \leq k)$	higher-order

Note that the triplets detailed in this table (and expanded below) are the typing rules that Coq obeys for determining the types of the dependent product. The choice of these triplets can affect the consistency of the type system, so have to be chosen with some care when designing type systems.

We will go through each row of this table in turn, giving examples only for the more commonly used rules.

2.2.6.1 Simple Types

The following constructs are the possible kinds of simple types:

$$\frac{\texttt{A: Set} \qquad \texttt{a: A} \vdash \texttt{B: Set}}{\texttt{forall a: A, B: Set}}$$

An example of this type is the type of a function `empty_list` that returns a list of a given length:

```
Parameter list_of_length: nat → Set.
Parameter empty_list: forall n, list_of_length n.
```

Due to this rule, the type of the type of `empty_list` is `Set`.

$$\frac{\texttt{A: Prop} \qquad \texttt{a: A} \vdash \texttt{B: Prop}}{\texttt{forall a: A, B: Prop}}$$

$$\frac{\texttt{A: Set} \qquad \texttt{a: A} \vdash \texttt{B: Prop}}{\texttt{forall a: A, B: Prop}}$$

This rule allows the creation of propositions quantified over types belonging to `Set`. For example, `n` = 1 is of type `Prop`, and `nat` is of type `Set`, so `forall n: nat, n` = 1 is of type `Prop`. This rule is used very often when writing theorems.

$$\frac{\texttt{A: Prop} \qquad \texttt{a: A} \vdash \texttt{B: Set}}{\texttt{forall a: A, B: Set}}$$

2.2.6.2 Impredicativity in `Prop`

$$\frac{\texttt{A: Type(i)} \qquad \texttt{a: A} \vdash \texttt{B: Prop}}{\texttt{forall a: A, B: Prop}}$$

This rule allows the construction of propositions quantified over propositions (including themselves). This is an advanced feature of Coq and is not used in this report, but here is a short example usage of this rule:

```
Parameter false_impredicative_defn: forall P: Prop, P.
```

`false_impredicative_defn` states that all propositions hold. This is clearly false, unless we have a contradiction, so is an alternate way of defining `False` (how the `False` type works was mentioned in section 2.2.4). `false_impredicative_defn` itself is of type `Prop`, but is quantified over `Prop`, so its own proposition applies to itself. To put it another way, `false_impredicative_defn` asserts that all propositions are true, including itself.

2.2.6.3 Dependence

$$\frac{\texttt{A: Set} \qquad \texttt{a: A} \vdash \texttt{B: Type(i)}}{\texttt{forall a: A, B: Type(i)}}$$

This type `forall a: A, B: Type(i)` represents a family of types indexed by some set (eg the natural numbers).

$$\frac{\texttt{A: Prop} \qquad \texttt{a: A} \vdash \texttt{B: Type(i)}}{\texttt{forall a: A, B: Type(i)}}$$

2.2.6.4 Higher-order

$$\frac{\texttt{A: Type(i)} \qquad \texttt{a: A} \vdash \texttt{B: Type(j)}}{\texttt{forall a: A, B: Type(k)}}$$

This rule allows us to have *type constructors*. This is where we take existing types and use them to construct new types. An example of this is the `list` type, which has type `Type` $\rightarrow$ `Type`, which is a simple version of this rule where the quantified term `a` doesn't appear in `B`.

2.2.7 Law of Excluded Middle

In section 2.2.4, it was said that a proof of a proposition is an element which has the proposition as its type. This means that, in Coq, to prove a proposition, it is not allowed to simply prove that the negation of the proposition leads to a contradiction. Rather, a direct proof of the proposition must be constructed.

A consequence of this is that various mathematical theorems cannot be proved within Coq, which are considered provably true in classical logic. The fundamental example of this is the *law of excluded middle*:

```
Theorem p_or_not_p: forall P: Prop, (P ∨ ¬P).
(* Note that ∨ is the disjunction operator aka 'or' *)
```

In Coq, it is impossible to construct a proof of this theorem. Note that it is possible to construct a proof of the double-negation of this theorem (note that this proof and others in this section use tactics which are explained later in section 2.2.10):

```
Theorem law_of_excluded_middle: forall A: Prop, ¬¬(A ∨ ¬A).
Proof.
  intros A H.
  apply H.
  right.
  intros a.
  apply H.
  left.
  assumption.
Qed.
(* This constructs the proof:
fun (A : Prop) (H : ¬ (A ∨ ¬ A)) ⇒ H (or_intror (fun a : A ⇒ H
    (or_introl a)))
     : forall A : Prop, ¬ ¬ (A ∨ ¬ A)   *)
```

This then also indicates that the following theorem can not be proved in Coq:

```
Theorem double_neg: forall A: Prop, ¬¬A → A.
```

This theorem can be thought of as stating that if the negation of a theorem is false, then the theorem itself must be true. This is essentially proof by contradiction, a concept rejected by the constructive logic, *The Calculus of Constructions*, which Coq uses, as proof by contradiction does not directly construct a proof of a theorem, but instead shows that the converse leads to a contradiction.

We now have discussed 3 theorems:

1. `forall P: Prop, P ∨ ¬P`
2. `forall P: Prop, ¬¬P → P`
3. `forall P: Prop, (¬P → False) → P` (this wasn't mentioned explicitly above, but is proof by contradiction)

and have said that they are all equivalent. We can prove that all three of these theorems are equivalent using Coq (this proof uses the theorem `law_of_excluded_middle` we have just proven). We do the following proofs:

- theorem 2 implies theorem 1
- theorem 3 implies theorem 2
- theorem 1 implies theorem 3

```
Theorem two_implies_one: (forall P: Prop, (¬¬P → P)) → (forall P, (P ∨ ¬P)).
Proof.
  intros H P.
  apply H.
  exact (law_of_excluded_middle P).
Qed.
(* This constructs the proof:
fun (H : forall P : Prop, ¬ ¬ P → P) (P : Prop) ⇒ H (P ∨ ¬ P)
    (law_of_excluded_middle P)
     : (forall P : Prop, ¬ ¬ P → P) → forall P : Prop, P ∨ ¬ P *)

(* As ¬P in Coq is the same as P → False, this theorem is actually
   really straightforward to prove, as both sides of the implication
   are essentially the same statement *)
Theorem three_implies_two: (forall P: Prop, (¬P → False) → P) → (forall P:
    Prop, ¬¬P → P).
Proof.
  intro H.
  exact H.
Qed.
(* This constructs the proof:
 fun H : forall P : Prop, (¬ P → False) → P ⇒ H
     : (forall P : Prop, (¬ P → False) → P) → forall P : Prop, ¬ ¬
    P → P
```

```
which without the types is just:
fun H ⇒ H
which is just the identity function *)

Theorem one_implies_three: (forall P: Prop, P ∨ ¬P) → (forall P: Prop, (¬P →
    False) → P).
Proof.
  intros P_or_not_P P not_not_P.
  case (P_or_not_P P).
  - intro p. exact p.
  - intros not_p. apply False_ind.
    apply not_not_P.
    exact not_p.
Qed.
Print one_implies_three.
(* This constructs the proof:
fun (P_or_not_P : forall P : Prop, P ∨ ¬ P) (P : Prop) (not_not_P : ¬
    P → False) ⇒
match P_or_not_P P with
| or_introl p ⇒ p
| or_intror not_p ⇒ False_ind P (not_not_P not_p)
end
     : (forall P : Prop, P ∨ ¬ P) → forall P : Prop, (¬ P → False)
    → P *)
```

2.2.8 Inductive types

So far, we have discussed using types, but we haven't discussed how to create types. This section focusses on this, creating a form of types known as *inductive types* (we have seen other kinds of types already, such as *function types* and *dependent types*).

Inductive types are created using the `Inductive` command. A simple example of using this command is:

```
Inductive bool: Set :=
| true: bool
| false: bool.
```

This defines an inductive type called `bool`, of which there are only two elements with that type: `true` and `false`. The elements `true` and `false` are also known as the *constructors* of the type `bool` (the reason for this will become clearer later in

this section). `bool` itself has sort `Set`. This is in fact exactly how the type `bool` is defined in the standard libray (in file `theories/Init/Datatypes.v`[1]).

Like in section 2.1.5, we can take advantage of type inference in Coq and rewrite this definition as:

```
Inductive bool: Set :=
| true
| false.
```

or we can move everything onto one line:

```
Inductive bool: Set := true | false.
```

All of these definitions are equivalent, though it is often better to explicitly write down all types when defining inductive types to make the code more readable.

To illustrate some more details of Coq, we will cover the following example (the original exercise can be found in [1] p. 139):

```
Section exercises_6_1.
  Inductive month : Set :=
  | January  | February | March    | April
  | May      | June     | July     | August
  | September | October  | November | December.

  Inductive season : Set :=
  | Spring | Summer | Autumn | Winter.

  Definition month_to_season: month → season :=
    month_rec (fun _ ⇒ season)
      Winter Winter Winter
      Spring Spring Spring
      Summer Summer Summer
      Autumn Autumn Autumn.
End exercises_6_1.
```

First we define two inductive types, `month` and `season`, in the same way that the `bool` type was defined (here we didn't write down all the types explicitly as that would have made the code less readable due to the number of constructors).

Whenever an inductive type is created, such as `month`, Coq automatically creates the following variables with the following types:

[1] See the url https://github.com/coq/coq/blob/859a9666923e657add7e972762af29a1872cc842/theories/Init/Datatypes.v#L33-L35.

```
month_rec: forall P : forall P : month → Set,
 P January →
 P February →
 P March → P April → P May → P June → P July → P August → P
   September → P October → P November → P December → forall m : month,
   P m
```

```
month_ind: forall P : forall P : month → Prop,
 P January →
 P February →
 P March → P April → P May → P June → P July → P August → P
   September → P October → P November → P December → forall m : month,
   P m
```

```
month_rect: forall P : month → Type,
 P January →
 P February →
 P March → P April → P May → P June → P July → P August → P
   September → P October → P November → P December → forall m : month,
   P m
```

`month_rec` is a function (with a dependent type) that takes as input a function `P` of type `month → Set` (so this function associates a type of sort `Set` to each constructor of `month`), values for each of the months that are of the correct type as per the function `P`, and returns a dependently typed function that takes as input any month and returns the corresponding value. The function `month_to_season` in the example above shows this function `month_rec` being used in this way, and may make this explanation make more sense, as the type `month_rec` is very general.

`month_ind` does exactly the same thing as `month_rec`, except whereas `month_rec` takes as input a function of type `month → Set`, `month_ind` takes as inpt a function of type `month → Prop` (ie mapping months to propositions). Then the remaining inputs to `month_ind` are proofs of the corresponding propositions.

`month_rect` is also very similar to `month_rec` and `month_ind`, but takes as input a function of type `month → Type`.

`month_rec` can be used to construct a function that operates over all elements of an inductive type, whilst `month_ind` can be used to construct proofs of propositions that are quantified over all elements of an inductive type.

Closely related to inductive types is pattern matching. This provides a convenient way of defining functions (among other uses). For example:

```
Section exercises_6_5.
  Definition month_length_is_even := fun (leap: bool) (m: month) ⇒
    match m with
    | February ⇒ if leap then false else true
    | April ⇒ true
    | June ⇒ true
    | September ⇒ true
    | November ⇒ true
    | _ ⇒ false
    end.
End exercises_6_5.
```

Here we are using the `month` inductive type defined above. This function `month_length_is_even` takes as input a month and a boolean indicating whether or not it is a leap year, and returns whether or not the number of days in the month is an even number. What is important to note is that this function uses a `match` clause. The variable `m` is compared to each statement in order (`February`, `April` and so on) and if it matches that statement then the expresssion after the ⇒ is evaluated and returned. The final statement `m` can be matched against is `_` which matches all remaining forms of `m` that haven't been matched by any of the previous statements.

The first statement for `February` returns `false` or `true` depending on whether `leap` is `true` or `false` respectively. All the other statements return `true` or `false` as in the function.

This function `month_length_is_even` could have been defined using `month_rec` instead:

```
Definition month_length_is_even' (leap: bool) :=
  month_rec (fun _ ⇒ bool)
            false (if leap then false else true) false true
            false true false false
            true false true false.
```

In fact, `month_rec` is just defined using `month_rect` (as is `month_ind`):

```
Definition month_rec := fun P : month → Set ⇒ month_rect P.
```

and `month_rect` is defined as:

```
Definition month_rect :=
fun (P : month → Type) (f : P January) (f0 : P February)
  (f1 : P March) (f2 : P April) (f3 : P May) (f4 : P June)
  (f5 : P July) (f6 : P August) (f7 : P September)
  (f8 : P October) (f9 : P November) (f10 : P December)
  (m : month) ⇒
match m as m0 return (P m0) with
| January ⇒ f
| February ⇒ f0
| March ⇒ f1
| April ⇒ f2
| May ⇒ f3
| June ⇒ f4
| July ⇒ f5
| August ⇒ f6
| September ⇒ f7
| October ⇒ f8
| November ⇒ f9
| December ⇒ f10
end.
```

so `month_rect` itself is defined using pattern matching. Note that the part of the definition `match m as m0 return (P m0) with` handles the function `P` potentially returning different types for different months (in the examples above we have restricted ourselves to returning the same type in all cases).

Note that pattern matching in Coq is similar to that in other functional programming languages, such as Haskell (although Haskell has a simpler version of pattern matching due to the lack of dependent types).

2.2.8.1 Inductive types with constructors accepting parameters

So far, all the constructors of the inductive types we have created have accepted no parameters, but this need not be the case. To illustrate this, let us take a look at the definition of the `nat` inductive type in the standard library[2]:

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

This type definition distinguishes between two types of natural numbers:

[2] See the url https://github.com/coq/coq/blob/859a9666923e657add7e972762af29a1872cc842/theories/Init/Datatypes.v#L134-L136.

- zero, denoted by the letter `O`
- successor numbers, which are constructed using the constructor `S`, which is a function that takes as input a natural number and returns a new natural number (representing the original natural number + 1)

Examples of elements of type `nat` are:

Element	Number it represents
O	0
S O	1
S (S O)	2
S (S (S O))	3
...	...

Coq permits using the numerals to represent elements of the type `nat`, so instead of writing `S O` we can write 1. This also works for other number types in the standard library.

An important point to note about this example is that there exists an infinite number of elements of type `nat`, due to the recursive definition of the type.

We will now take a look at two other types from the standard library[3]:

```
Inductive positive : Set :=
  | xI : positive → positive
  | xO : positive → positive
  | xH : positive.

Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

The type of interest is the `Z` type, but as it uses the `positive` type, we'll briefly cover that first.

The `positive` type represents binary numbers that are >=1. The constructors work as follows:

- `xH` represents 1
- `xO` represents taking a number and multiplying it by 2

[3] See the url https://github.com/coq/coq/blob/859a9666923e657add7e972762af29a1872cc842/theories/Numbers/BinNums.v#L21-L24 and https://github.com/coq/coq/blob/859a9666923e657add7e972762af29a1872cc842/theories/Numbers/BinNums.v#L51-L54.

- `xI` represents taking a number, multiplying it by 2 and then adding 1

So, for example, 13 is represented as `xI (xO (xI xH))`. An alternative way of understanding this is to write 13 in binary as 1101, reverse it to get 1011, then the following table shows how the constructors map to each digit in order:

1	0	1	1
`xI`	`xO`	`xI`	`xH`

Then, the `Z` type is defined using the `positive` type. The constructors of the `Z` type work as follows:

- `Z0` represents 1
- `Zpos` represents a positive number
- `Zneg` represents a negative number

What is important to note is that constructors of inductive types can take parameters of arbitrary types as input (in this case, `positive`).

Inductive types can be parametrised. A good example of this is the `list` type, which is defined in the standard library as follows[4]:

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

This defines a `list` type that contains objects of the same type. We can either have an empty list by using the `nil` constructor, or we can take a list and prepend an element to it to get a new list, using the `cons` constructor. Note that the `cons` constructor can also be shortened to `::` in the scope `list_scope`, so the following are equivalent:

- `(cons 1 (cons 2 nil))`
- `(1 :: 2 :: nil)%list`

Inductive types can also be used to construct propositions. An example of this is the `le` type, defined as:

```
Inductive le (n: nat): nat → Prop :=
| le_n: le n n
| le_S: forall m: nat, le n m → le n (S m).
```

[4] see the url https://github.com/coq/coq/blob/859a9666923e657add7e972762af29a1872cc842/theories/Init/Datatypes.v#L224-L226

The `le` type is used to build propositions representing the $\leq$ relation. This type has several parts to it:

- In the first line, the code (`n: nat`): `nat → Prop` tells Coq that `le` takes two numbers (one of which is explicitly named `n` at the top of the definition and then referenced by both constructors) and returns a proposition. The `le` type could equivalently have been written as:

```
Inductive le: nat → nat → Prop :=
 | le_n: forall n: nat, le n n
 | le_S: forall n m: nat, le n m → le n (S m).
```

  but this definition repeats `forall n` in both constructors.

- The constructor `le_n` gives us a term of type `le n n` for any number `n`. This represents the assertion that $\forall n, n \leq n$.
- The constructor `le_S`, takes a term of type `le n m` and returns a term of type `le n (S m)`. In other words, given that $n \leq m$, we have $n \leq m + 1$.

2.2.9 Recursive functions

In most programming languages, functions are permitted to invoke themselves. However, this has the potential to allow functions to not terminate in a finite amount of time, something which is not allowed in Coq. One of the classic theorems in computability theory is the halting problem, which can roughly described as follows:

> Is it possible to create a function which, given an arbitrary program written in a Turing-complete language (where the program is encoded in a suitable form), outputs 0 if the program halts in a finite amount of time and 1 otherwise?

The answer is no (a proof is not provided here). Thus, the Coq interpreter cannot enforce that functions in Coq always terminate without excluding some functions that do terminate. This means that Coq is limited in its expressiveness. Another limit to the power of Coq was discussed in section 2.2.7.

Recursive functions can be defined in Coq, but the Coq interpreter must be able to tell that the recursive function always terminates. It does this by seeing that one of the parameters of the function gets simplified with each recursive call.

The `Fixpoint` command is used to define recursive functions. An example of its usage is as follows:

```
Section exercises_6_16.
  Fixpoint add_in_2nd_arg (a b: nat){struct b}: nat :=
    match b with
```

```
    | 0 ⇒ a
    | S b' ⇒ S (add_in_2nd_arg a b')
  end.
End exercises_6_16.
```

`add_in_2nd_arg` takes as input two numbers and adds them together. Pattern matching is done on the second input number, and the recursive call is always done with the second argument having been simplified (by having one `S` constructor removed, equivalent to subtracting 1). There is also a base case for when the second input number is 0. In this case, as `forall a: nat, a + 0 = a`, we just return `a`, the first input number.

In this example, we have used a bit of new syntax: `{struct b}`. This tells Coq that this function always decreases/simplifies the `b` parameter on each recursive call.

We can also define anonymous recursive functions. This is done using the `fix` syntax. An example of this is as follows:

```
Section exercises_6_38.
  (* Build a function that takes a number n as input and returns a list
     of consecutive numbers from 1 to n inclusive, in that order. *)
  Definition list_consecutive_upto_n (n: nat) :=
    (fix f m acc :=
       match m with
       | 0 ⇒ acc
       | S m' ⇒ f m' (m :: acc)%list
       end) n nil.
End exercises_6_38.
```

This example uses the `list` inductive type, which was introduced in section 2.2.8.1.

The function `list_consecutive_upto_n` takes as input a number `n` and then invokes an anonymous recursive function with that number and an empty list. The anonymous function recursively decreases the input number whilst prepending numbers to the input list. Note that in the code `fix f m acc :=`, the variable `f` is set by the Coq interpreter to the anonymous function itself, so that it has a variable which it can call to invoke itself.

A more complex example involving recursive functions (both named and anonymous) and inductive types is given here (taken from [1] p. 172).

We first define the type `Z_inf_branch_tree`:

```
Section exercises_6_28.
  Inductive Z_inf_branch_tree: Set :=
    Z_inf_leaf: Z_inf_branch_tree
  | Z_inf_node: Z → (nat → Z_inf_branch_tree) → Z_inf_branch_tree.
```

This represents trees of integers (the tree data structure is found in many programming languages). We have two constructors for this type:

- `Z_inf_leaf` which represents an empty tree
- `Z_inf_node` which constructs a tree by taking an integer and a function that generates child trees.

The trees this type represents have an infinite number of branches. This is due to providing a function to the `Z_inf_node` constructor, where the function can be thought of as representing an infinite number of branches indexed by a natural number.

This is not an easy type to understand, so to give a better idea of what terms of this type look like, figure 2.2 shows graphically how such terms look.

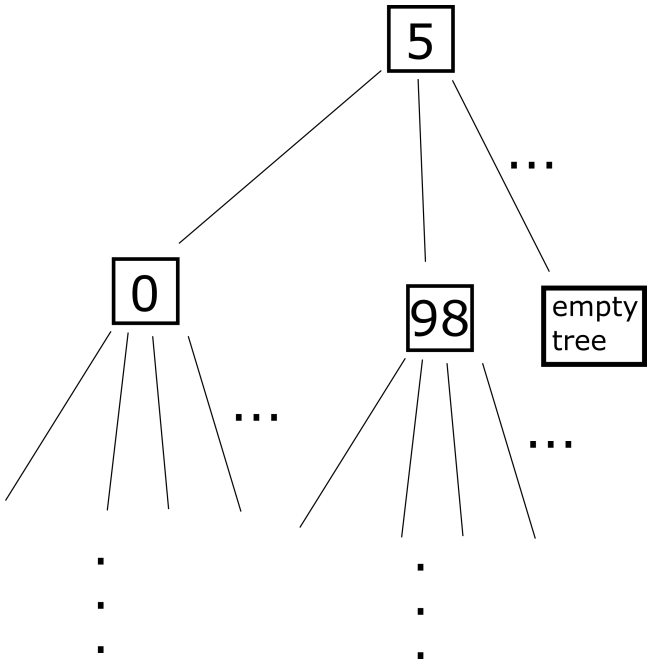

Figure 2.2: Graphical representation of `Z_inf_branch_tree`

Now that we have a type, we will define a recursive function that operates

over this type. As recursion in Coq must always terminate, and we have a tree with infinite branches (where each branch can itself be a tree with infinite branches), our function must somehow be restricted to operate only over a subset of branches at each level of the tree.

The function we define, `zero_in_tree`, takes as input a tree `tree` and a natural number `n`. It recursively searches through the tree to see if the number 0 is in the tree. The search is restricted to branches indexed by natural numbers upto and including `n`.

```
  Fixpoint zero_in_tree (n: nat) (tree: Z_inf_branch_tree): bool :=
2   match tree with
    | Z_inf_leaf ⇒ false
4   | Z_inf_node 0%Z _ ⇒ true
    | Z_inf_node _ f ⇒ match n with
6                      | 0 ⇒ zero_in_tree n (f 0)
                       | _ ⇒ (fix g (n': nat): bool :=
8                               match n' with
                                | 0 ⇒ zero_in_tree n (f 0)
10                              | S n'' ⇒ orb (zero_in_tree n (f n')) (g n'')
                                end ) n
12                     end
    end.
14 End exercises_6_28.
```

Note that the `orb` function used here is defined in the standard library as:

```
Definition orb: = fun b1 b2 : bool ⇒ if b1 then true else b2.
```

that is, it returns the boolean OR of its two inputs (which are booleans).

`zero_in_tree` is quite a complex function (and was not trivial to create). It works as follows:

1. If our tree is an empty tree, return `false`.
2. If our tree has 0 at its root node, then return `true`.
3. Otherwise, for each branch i where $i \leq n$, invoke the `zero_in_tree` function on that branch, returning `true` if any of these function invocations returns `true`, and returning `false` if none of them return `true`.

2.2.10 Tactics

Tactics are one of the main strengths of Coq. They are primarily used to prove theorems. Essentially, they provide a more intuitive and interactive way to construct proof terms for a given theorem. To give a quick example of their power, here is a proof done both with and without tactics:

```
Section Exercises_5_6.
  Theorem conj_dist: forall A B C: Prop, A ∧ (B ∧ C) → (A ∧ B) ∧ C.
  Proof.
    intros A B C H.
    elim H.
    intros a b_and_c.
    elim b_and_c.
    split. split.
    assumption. assumption. assumption.
  Qed.

  Theorem conj_dist': forall A B C : Prop, A ∧ B ∧ C → (A ∧ B) ∧ C.
  Proof (fun (A B C : Prop) (H : A ∧ B ∧ C) ⇒
    and_ind
      (fun (a : A) (b_and_c : B ∧ C) ⇒
       and_ind (fun (H0 : B) (H1 : C) ⇒ conj (conj a H0) H1) b_and_c) H).
End Exercises_5_6.
```

The second proof doesn't use tactics, whereas the first one does. Without trying to understand how the proof itself works, the first proof looks a lot more readable than the second one. The first proof also is better structured with whitespace to make it easier to read.

The Coq standard library comes with many built-in tactics, and Coq allows for creating new tactics (this is not used or covered in this report).

2.2.10.1 How to use tactics

Before giving examples of the different tactics available in Coq, we will work through a simple example to see how to use them. Coq is designed for interactive use, and this is especially true of tactics. Thus, the reader is encouraged to work through the following example on their own computer, in order to get a better understanding.

The proposition we will prove is taken from [1] p. 93. We first start by declaring some variables:

```
Variables (A: Set) (P Q: A → Prop).
```

The proposition we want to prove is (`forall a: A, P a → Q a) → (forall a: A, P a) → forall a: A, Q a`. We use the `Theorem` command, which we use as follows:

```
Theorem all_imp_dist: (forall a: A, P a → Q a) → (forall a: A, P a) → forall
    a: A, Q a.
```

We have given the name `all_imp_dist` to our theorem. More complicated proofs can use proofs of other theorems, so choosing good names for theorems helps make proofs more readable.

The `Theorem` command we just entered produces the following output:

```
1 subgoal
2 ____________________________________(1/1)
(forall a : A, P a → Q a) → (forall a : A, P a) → forall a : A, Q a
```

The Coq interpreter has now entered its *interactive proof mode*. We can see our proposition, and we are told that there is only one subgoal to prove, namely our original proposition. In proof mode, we start with the proposition we want to prove and work backwards through the proof until there is nothing more to prove.

To start our proof, we use the `Proof` command:

```
Proof.
```

This isn't strictly necessary, but makes the code more readable, so it is nearly always used in practice.

We will now use the `intros` tactic:

```
intros H0 H1 a.
```

This gives us the following output:

```
1 subgoal
H0 : forall a : A, P a → Q a
3 H1 : forall a : A, P a
a : A
5 ____________________________________(1/1)
Q a
```

We still have one subgoal, but this is now `Q a`, which is a lot simpler than the original proposition. What we have also done is introduced three *hypotheses* to the *context*, which we have also named `H0`, `H1` and `a`. These new hypotheses have come from the previous subgoal as per this table:

H0		H1		a	our new subgoal
(forall a : A, P a -> Q a)	->	(forall a : A, P a)	-> forall	a : A,	Q a

Essentially, the idea behind the `intros` tactic is that it can take a goal that of the form `A → B` and turn `A` into a hypothesis and `B` into the new goal. The `intros` tactic can do this for multiple hypotheses at a time.

So, our subgoal is `Q a` (ie we are now trying to prove that `Q a` holds), and looking at the hypotheses, we see that `H0` has the type `forall a : A, P a → Q a` which includes `Q a` at the end. We can now use the `apply` tactic with this hypothesis to transform our subgoal:

```
apply (H0 a).
```

This gives us the output

```
1 subgoal
H0 : forall a : A, P a → Q a
H1 : forall a : A, P a
a : A
_____________________________________(1/1)
P a
```

What we passed to the `apply` tactic was `H0 a`, which has the type `P a → Q a`. The apply tactic matched the part after the `→` with the subgoal and replaced the subgoal with the part before the `→`. Thus our subgoal is now `P a`.

Now we can see that this looks very much like the hypothesis `H1`. So we can again use the `apply` tactic:

```
apply (H1 a).
```

This gives us the output `No more subgoals.`. As soon as we get this output, we have proved our theorem. To generate the proof term for our theorem and make our proved theorem available as a variable, we use the `Qed` command:

```
Qed.
```

Now we have left the proof mode. Here is our proof in its entirety:

```
Variables (A: Set) (P Q: A → Prop).
Theorem all_imp_dist: (forall a: A, P a → Q a) → (forall a: A, P a) → forall
    a: A, Q a.
Proof.
intros H0 H1 a. apply (H0 a). apply (H1 a).
Qed.
Print all_imp_dist.
```

We can take a look at the proof term we constructed using tactics by entering `Print all_imp_dist.`:

```
all_imp_dist =
fun (H0 : forall a : A, P a → Q a) (H1 : forall a : A, P a) (a : A) ⇒
H0 a (H1 a)
     : (forall a : A, P a → Q a) → (forall a : A, P a) → forall a : A, Q a

Argument scopes are [function_scope function_scope _]
```

Getting rid of the types in this definition leaves us with:

```
all_imp_dist = fun H0 H1 a ⇒ H0 a (H1 a)
```

Comparing this to our proof, we can see that `intros` corresponds to function parameters and `apply` corresponds to invoking functions.

Note that in this example, we only ever had at most one subgoal. In general, some tactics can create multiple subgoals. In this case, you simply solve each subgoal in turn. No extra syntax is required, although later in this section we will see some syntax that can make such proofs look neater.

All proofs in Coq work in roughly the same way as this one:

- State the theorem to be proved using the `Theorem` command (or a synonym such as `Lemma` or `Remark`)
- Use a sequence of tactics to simplify the subgoal until it is proved
- Type the `Qed` command to finish the proof

2.2.10.2 Individual tactics

There are many tactics in the Coq standard library, and many more in external libraries that people have created (Coq also contains commands that allow creating new tactics). We stick to the simplest ones, although even these provide more than enough power to tackle complicated proofs, if used correctly.

In this section, we will go through all the tactics used in section 3 in turn, with examples. A lot more detail about all these tactics and more can be found in [2].

2.2.10.2.1 exact

The `exact` tactic allows you to just manually enter a proof term for the current goal instead of using tactics. Sometimes, this can be useful when the goal is simple enough. For example:

```
Section Exercises_5_6.
  Theorem not_true_and_false: forall A: Prop, ¬(A ∧ ¬A).
    intros A H.
    elim H.
    intros a neg_a.
```

At this point, the state of the proof looks like:

```
1 subgoals
A : Prop
H : A ∧ ¬ A
a : A
neg_a : ¬ A
______________________________________(1/1)
False
```

We can use tactics to finish this proof, or we can use the `exact` tactic:

```
    exact (neg_a a).
  Qed.
End Exercises_5_6.
```

`neg_a` has type `¬A` which is shorthand for `A → False`. So `neg_a` is a function that, if given an element of type `A` (`a` is a hypothesis with such a type), will give us an element of type `False`, which is what we want.

2.2.10.2.2 intro/intros

If we have a goal of the form `A1 → A2 → A3 → ... → An` and we use the `intro` tactic, then the goal is transformed into `A2 → A3 → ... → An` and `A1` is added to the context as a hypothesis. `intro` can be passed a name which will be the name given to the new hypothesis (eg `intro name_here.`).

The `intros` tactic is simply a shorthand for repeatedly invoking the `intro` tactic. Again, the `intros` tactic can be passed names which will be used to set the

names of the newly created hypotheses. So again, if we have a goal of the form `A1 → A2 → A3 → ... → An` and we enter `intros a b.` into the Coq interpreter, then the goal is transformed into `A3 → ... → An` and we get two new hypotheses: `a: A1` and `b: A2`.

Note that both the `intro` and the `intros` tactic don't need to be passed the name(s) of the new hypotheses. When this is done, then the names are chosen automatically. This makes Coq source code unreadable, so should be avoided when sharing code. Also note that calling `intros` with no names is equivalent to repeatedly calling `intro` as many times as possible. So, running `intros.` on a goal `A1 → A2 → A3 → ... → An` would give a goal `An` and $n-1$ new hypotheses.

2.2.10.2.3 apply

If we have a goal of the form `An` and there exists a variable or hypothesis `a: A1 → A2 → ... → An`, then running the tactic `apply a.` generates $n-1$ new goals which would be `A1`, `A2`, ..., `An`-1. To explain this better, we will go through the following example:

```
Variables (P Q R: Prop).

Theorem apply_example: (P → Q → R) → P → Q → R.
Proof.
  intros f p q.
```

At this point, we have the following output:

```
1 subgoal
f : P → Q → R
p : P
q : Q
_____________________________________(1/1)
R
```

`f` is of the type `P → Q → R`, and the last part of this matches the current subgoal, so we use it with `apply`:

```
  apply f.
```

Now we get the output:

```
2 subgoals
f : P → Q → R
p : P
```

```
q : Q
_____________________________________(1/2)
P
_____________________________________(2/2)
Q
```

We now have two subgoals. Each of these can be solved trivially using the hypotheses `p` and `q` for the first and second subgoal respectively:

```
  - exact p.
  - exact q.
Qed.
```

Note that we used the syntax – here. This isn't necessary, but makes it clear when one is solving a particular subgoal.

2.2.10.2.4 simpl

The `simpl` tactic simplifies the current goal. What this can entail is attempting to evaluate function calls and performing pattern matching. This tactic is best explained by seeing an example:

```
Require Import ZArith.
Section exercises_6_20.
  Fixpoint pos_even_bool (p: positive): bool :=
    (match p with
     | xO p' ⇒ true
     | other ⇒ false
     end)%positive.

  Lemma pos_even_bool_returns_true_for_even_nums:
    forall p: positive, pos_even_bool (2 * p) = true.
  Proof.
    intros p.
```

We imported `ZArith` to use the binary `positive` numbers. We then define the function `pos_even_bool` which returns true if the input number is even and false otherwise (the way the `positive` type is defined makes this straightforward, see section 2.2.8.1 for more details about how the `positive` type works).

We then start proving the theorem `pos_even_bool (2 * p) = true.`. The left-hand side of this expression can be simplified, and we can do this using the `simpl` tactic:

```
    simpl.
```

Now the state of our proof is:

```
1 subgoal
p : positive
_____________________________________(1/1)
true = true
```

What the `simpl` tactic did was it first simplified `2 * p` to `xO p` by looking at the definition of the `mul` function (which we won't cover here). It then looked at `pos_even_bool`, and using pattern matching, identified that `xO p` matches the first clause of the match statement in the function. Thus, the right-hand side of this statement, `true` was convertible with `pos_even_bool (xO p)`, and so the final goal we ended up with was `true = true`.

Now we can finish off our proof:

```
    reflexivity.
  Qed.
End exercises_6_20.
```

2.2.10.2.5 reflexivity

The `reflexivity` tactic operates on goals of the form `a = b`. If, for some proposition `a`, we have the goal `a = a`, then the `reflexivity` tactic will finish our proof:

```
Theorem equals_reflexive: forall A: Prop, A = A.
Proof.
  intro A.
  reflexivity.
Qed.
```

The `reflexivity` tactic will try and simplify both sides of the equality, such as in this example:

```
Section exercises_6_15.
  Definition nat_smaller_than_3 (n: nat): bool :=
    match n with
    | 0 | S 0 | S (S 0) ⇒ true
```

```
    | other ⇒ false
6   end.

8   Lemma nat_smaller_than_3_works_correctly_on_0:
      nat_smaller_than_3 0 = true.
10  Proof. reflexivity. Qed.

12  Lemma nat_smaller_than_3_works_correctly_on_1:
      nat_smaller_than_3 1 = true.
14  Proof. reflexivity. Qed.

16  Lemma nat_smaller_than_3_works_correctly_on_2:
      nat_smaller_than_3 2 = true.
18  Proof. reflexivity. Qed.

20 End exercises_6_15.
```

In all three of these lemmas, the left-hand side of the equality is simplified to `true`, and then the equality is verified to hold.

2.2.10.2.6 unfold

The `unfold` tactic unfolds a given variable in the current goal. By unfold, we mean that usages of the variable are replaced with the definition of the variable. For example:

```
(* Exercise 5.3 *)
2 Section Exercises_5_3.
    Theorem not_false: ¬False.
4   Proof.
```

`¬False` is shorthand for `not False`, where `not` is the function:

```
not = fun A : Prop ⇒ A → False
2     : Prop → Prop
```

We use the `unfold` tactic:

```
    unfold not.
```

We now get the output:

```
1 subgoal
_____________________________________(1/1)
False → False
```

This is now fairly straightforward to prove:

```
    intros f. exact f.
  Qed.
End Exercises_5_3.
```

2.2.10.2.7 assumption

When our goal is the same as one of our hypotheses, until now we have been using the `exact` tactic. We can alternatively use the `assumption` tactic which looks through the hypotheses, and if it finds one that matches the current goal, finishes the proof of the goal for us. For example:

```
(* Exercise 5.6 *)
Section Exercises_5_6.
  Theorem conj_dist: forall A B C: Prop, A ∧ (B ∧ C) → (A ∧ B) ∧ C.
  Proof.
    intros A B C H.
    elim H.
    intros a b_and_c.
    elim b_and_c.
    split. split.
```

At this point in the proof, we have the following output:

```
3 subgoals
A, B, C : Prop
H : A ∧ B ∧ C
a : A
b_and_c : B ∧ C
H0 : B
H1 : C
_____________________________________(1/3)
A
_____________________________________(2/3)
B
_____________________________________(3/3)
C
```

We could finish this proof with:

```
    exact a. exact H0. exact H1.
  Qed.
End Exercises_5_6.
```

or, using `assumption`:

```
    assumption. assumption. assumption.
  Qed.
End Exercises_5_6.
```

2.2.10.2.8 rewrite

The `rewrite` tactic takes a term of type `t = u` where `t` and `u` are propositions, and replaces all terms in the current goal that match `t` with `u`. For example:

```
Variables (P Q R: Prop).
Theorem rewrite_transitive_example: (P = Q) → (Q = R) → (P = R).
Proof.
  intros p_eq_q q_eq_r.
```

The state of the proof is:

```
1 subgoal
p_eq_q : P = Q
q_eq_r : Q = R
_____________________________________(1/1)
P = R
```

We can use the `rewrite` tactic with the hypothesis `p_eq_q` to replaces occurrences of `P` with `Q` in the goal:

```
  rewrite p_eq_q.
```

We now have a goal `Q = R`, so can use the `rewrite` tactic with the hypothesis `q_eq_r` to replaces occurrences of `Q` with `R`:

```
  rewrite q_eq_r.
```

Now our goal is `R = R` which is easy to prove:

```
  reflexivity.
Qed.
```

The `rewrite` tactic can also rewrite from right-to-left instead of left-to-right, by using `rewrite ← term.` instead of `rewrite term.`. Here is an alternative way of solving the above example:

```
Theorem rewrite_transitive_example': (P = Q) → (Q = R) → (P = R).
Proof.
  intros p_eq_q q_eq_r.
  rewrite ← q_eq_r.
  rewrite ← p_eq_q.
  reflexivity.
Qed.
```

2.2.10.2.9 pattern

Sometimes when using the `rewrite` tactic, we only want to rewrite specific expressions. The `rewrite` tactic finds the first term from the left-hand side of the goal that can be rewritten, and then rewrites all occurrences of that term in the goal. To rewrite more selectively, we can use the `pattern` tactic. This tactic takes as input the expression that will be rewritten, the number of occurrences that will be rewritten from left-to-right, and transforms the goal into one that uses function application on the target expression. This is best explained using an example:

```
(* Exercise 5.10 *)
Section Exercises_5_10.
  Require Import Arith.
  Theorem plus_permute2: forall n m p: nat, n+m+p = n+p+m.
  Proof.
    intros n m p.
```

In this example, we are given the following theorems from the standard library:

```
plus_assoc: forall n m p : nat, n + (m + p) = n + m + p
plus_comm: forall n m : nat, n + m = m + n
```

Our goal is currently `n + m + p = n + p + m`. First, we'll rewrite the addition on the right-hand side of the equality in the goal. To do this, we first use the `pattern` tactic:

```
    pattern (n + p + m) at 1.
```

which transforms our goal to `(fun n0 : nat ⇒ n + m + p = n0) (n + p + m)`. This may look strange, but it is convertible to the original goal. This new goal has our target expression `n + p + m` separated from everything else. Now we can do the rewrite:

```
    rewrite ← plus_assoc.
```

which transforms our goal to `n + m + p = n + (p + m)`. Using the `pattern` and `rewrite` tactics a bit more we can finish this proof:

```
    pattern (p + m) at 1. rewrite plus_comm.
    rewrite plus_assoc.
    reflexivity.
  Qed.
End Exercises_5_10.
```

2.2.10.2.10 induction, elim and destruct

Proof by induction can be done without tactics using the induction theorems generated when an inductive type is defined (eg `nat_ind`, `bool_ind`). To do such proofs using tactics, we can use the `induction` tactic. Here is an example:

```
Section exercises_6_29.
  Theorem plus_n_0: forall n: nat, n = n + 0.
  Proof.
```

To prove this theorem, we will need to do a proof by induction on the number `n`:

```
    intro n.
    induction n.
```

We now get the following output:

```
2 subgoals
_____________________________________(1/2)
0 = 0 + 0
_____________________________________(2/2)
S n = S n + 0
```

The first subgoal is straightforward to prove:

```
    - reflexivity.
```

The second subgoal requires using the induction hypothesis. We start with the following output:

```
1 subgoal
n : nat
IHn : n = n + 0
_____________________________________(1/1)
S n = S n + 0
```

The `induction` tactic has introduced the induction hypothesis into the context for us, and given it the name `IHn`. We first simplify our subgoal:

```
    - simpl.
```

which transforms our subgoal to `S n = S (n + 0)`. We can see that `n + 0` in the subgoal is also on the right-hand side of the equality in `IHn`, so we can use the `rewrite` tactic:

```
      rewrite ← IHn.
```

Our subgoal now is `S n = S n` which can be easily solved:

```
      reflexivity.
  Qed.
End exercises_6_29.
```

In this example, the `induction` tactic introduced hypotheses to the context with automatically-generated names. We could have, instead of using `induction n`.

in the example above, used `induction n as [|n IHn].` which would have made explicit the name of the induction hypothesis, which makes the proof script more readable.

There is a more basic but related tactic called `elim`. This tactic also can be used to do proof by induction, but it does not modify the context like the `induction` tactic does. We can repeat our example proof using the `elim` tactic:

```
Section exercises_6_29.
  Theorem plus_n_0: forall n: nat, n = n + 0.
  Proof.
    intro n.
    elim n.
```

The current state of the proof is now:

```
2 subgoals
n : nat
_____________________________________(1/2)
0 = 0 + 0
_____________________________________(2/2)
forall n0 : nat, n0 = n0 + 0 → S n0 = S n0 + 0
```

Notice that the second subgoal includes the induction hypothesis within the subgoal, instead of it being automatically added to the context. We can finish this proof in a similar manner to the first one:

```
    - reflexivity.
    - intros n' IHn. simpl.
      rewrite ← IHn.
      reflexivity.
  Qed.
End exercises_6_29.
```

There also exists a tactic called `destruct`. This tactic behaves similarly to `induction`. However, unlike `induction` or `elim`, the `destruct` tactic does not generate any induction hypotheses. In general, the `induction` and `elim` tactics will be used.

2.2.10.2.11 right, left and split

In section 2.2.5, the `and` and `or` functions were mentioned (along with their respective shorthand syntax $\wedge$ and $\vee$) as being used to construct propositions containing conjunctions and disjunctions (eg $(1 = 3 \vee 1 = 1) \wedge 5 = 5$). To do

proofs involving conjunctions and disjunctions, we can use a combination of the tactics `right`, `left`, `split` and `conj`.

We'll first look at disjunctions, which we'll explain with an example. Suppose we want to prove the following theorem:

```
(* Exercise 5.5 *)
Section Exercises_5_5.
  Theorem a_b_c_d_or: forall (A: Set) (a b c d: A), a=c ∨ b=c ∨ c=c ∨ d=c.
```

We'll start our proof in the usual way, introducing all quantified variables:

```
  Proof.
    intros A a b c d.
```

Now our goal is `a=c ∨ b=c ∨ c=c ∨ d=c`. If we can simplify this to `c=c`, then we can easily prove this goal. The `right` and `left` tactics allow us to do this. Running the `right` tactic:

```
    right.
```

changes our goal to `b = c ∨ c = c ∨ d = c`. Using `right` and `left` a few more times:

```
    right. left.
```

gets us to the goal `c=c`. We can now easily finish our proof:

```
    reflexivity.
  Qed.
End Exercises_5_5.
```

Running `Print a_b_c_d_or.`, we can see that our proof term we have constructed is:

```
a_b_c_d_or =
fun (A : Set) (a b c d : A) ⇒
or_intror (or_intror (or_introl eq_refl))
     : forall (A : Set) (a b c d : A),
       a = c ∨ b = c ∨ c = c ∨ d = c
```

This indicates to us that the `right` and `left` tactics are essentially shorthand for `apply or_intror.` and `apply or_introl.` respectively, where `or_intror` and `or_introl` are propositions with the types:

```
or_intror: forall A B : Prop, B → A ∨ B
or_introl: forall A B : Prop, A → A ∨ B
```

Now we'll look at conjunctions. Again, we'll look at an example:

```
(* Exercise 5.6 *)
Section Exercises_5_6.
  Theorem conj_dist: forall A B C: Prop, A ∧ (B ∧ C) → (A ∧ B) ∧ C.
  Proof.
    intros A B C H.
```

At this stage our proof state looks like:

```
1 subgoal
A, B, C : Prop
H : A ∧ B ∧ C
______________________________________(1/1)
(A ∧ B) ∧ C
```

The hypothesis `H` has type `A ∧ B ∧ C`. We can break this up into separate hypotheses using the `elim` tactic:

```
    elim H.
```

This changes our goal to be `A → B ∧ C → (A ∧ B) ∧ C`. We can introduce `A` and `B ∧ C` as hypotheses, then run the `elim` tactic on `B ∧ C` to break it up into separate hypotheses:

```
    intros a b_and_c.
    elim b_and_c. intros b c.
```

Now our proof state is:

```
1 subgoal
A, B, C : Prop
H : A ∧ B ∧ C
```

```
a : A
b_and_c : B ∧ C
b : B
c : C
______________________________________(1/1)
(A ∧ B) ∧ C
```

We now have the hypotheses `a`, `b` and `c` in our context, but our goal is still made up of conjunctions. We can split it up into separate subgoals for each of its components using the `split` tactic:

```
    split.
```

This gives us two subgoals: `A ∧ B` and `C`. Another use of the `split` tactic will break up the first subgoal:

```
    split.
```

Now we have the subgoals `A`, `B` and `C`. As all of these subgoals are hypotheses, we can easily finish this proof:

```
    assumption. assumption. assumption.
  Qed.
End Exercises_5_6.
```

Again, if we print the proof term we constructed using `Print conj_dist.`, we get the following output:

```
conj_dist =
fun (A B C : Prop) (H : A ∧ B ∧ C) ⇒
and_ind
  (fun (a : A) (b_and_c : B ∧ C) ⇒
   and_ind
     (fun (b : B) (c : C) ⇒ conj (conj a b) c)
     b_and_c) H
     : forall A B C : Prop,
       A ∧ B ∧ C → (A ∧ B) ∧ C
```

Of note here is the `conj` function, which has the type `forall A B : Prop, A → B → A ∧ B`. The `split` tactic is essentially just shorthand for `apply split.`.

2.2.10.2.12 discriminate

Whenever we have a goal that looks like `a <> b`, where `a` and `b` are of the same inductive type, but built using different constructors, we can prove this goal using the `discriminate` tactic. A simple example of its usage is as follows:

```
Theorem zero_nequal_one : 0 <> S 0.
Proof.
  intro zero_equals_one.
  discriminate zero_equals_one.
Qed.
```

We will not use the `discriminate` tactic much.

2.2.10.2.13 change

Sometimes, the current goal can be rewritten in a different form that is convertible to the original goal, but is more useful for constructing a proof. The `change` tactic allows this to be done. The following example shows how the `change` tactic works.

Suppose for instance that we had a goal $1 = 1$:

```
Theorem change_example: 1 = 1 + 0.
Proof.
```

Using the `change` tactic, we can rewrite the goal in a simpler form

```
change (1 = 1).
```

Now our goal is $1 = 1$, which we can easily prove.

```
reflexivity.
Qed.
```

2.2.10.2.14 assert

Sometimes, in the middle of a complex proof, we may wish to prove a lemma, and then use that lemma in the proof. One way of doing this is to prove the lemma separately, and then reference it inside the proof. Another way is to embed the proof of the lemma inside the original proof. This means that the lemma will not be available once the original proof is completed. The `assert` tactic fulfils this role. Here is an example of its usage:

```
Theorem assert_example: forall (m n p q: nat), m = n → n = p → p = q → m =
    q.
Proof.
  intros m n p q.
  intros m_eq_n n_eq_p p_eq_q.
```

At this point, our goal is `m = q`. Instead of solving it in one go, we can solve `m = p` separately, then use this proof to solve the original goal:

```
  assert (H: m = p).
```

The proof state now looks like:

```
2 subgoals
m, n, p, q : nat
m_eq_n : m = n
n_eq_p : n = p
p_eq_q : p = q
______________________________________(1/2)
m = p
______________________________________(2/2)
m = q
```

We first prove that `m = p`:

```
  - rewrite m_eq_n. rewrite n_eq_p. reflexivity.
```

and can now prove our original goal `m = q`:

```
  - rewrite H. rewrite p_eq_q. reflexivity.
Qed.
```

Chapter 3

A library for natural numbers

This project was split into two parts: learning how to use Coq (to a basic level), and applying this knowledge to rewrite part of the standard library. In order to do this rewrite, I first needed to appraise the state of the standard library, and based on that do the rewrite.

3.1 Overview of the standard library

The Coq standard library is quite large, consisting of code spread across over 400 files in over 40 folders. For this project, I focussed solely on code relating to the natural numbers (ie the `nat` type).

I captured my study of the existing state of the standard library in [3]. Initially, I started by picking some files that contained code relating to natural numbers, and seeing which files they imported, and which files were imported by those files, and so on. For each of these files, I manually read through the source code, noting at a high level what functions each file served. Whilst going through this process, I documented the dependencies between different files using tags in the tiddlywiki document.

This method initially worked well, especially for relatively simple files, but over time ran into several limitations:

- There are a lot of files in the standard library, and even focussing just on theorems and functions related to natural numbers, there was a prohibitively large number of files to explore and understand.
- Many times, it was not clear where a particular variable was defined when reading through Coq source code. Just looking at which files were

imported was not enough, as variables can be defined many files away in the graph of dependencies. For example, if file A imports B which imports C which imports D, and a variable was used but not defined in A, then one would have to recursively check the files imported by A, then the files imported by those files, and so on. I was to some extent able to mitigate this by searching through all the files in the standard library for usages of a particular variable, but this didn't work well for variables that were commonly used.

An alternative approach I used for some of the modules with large amounts of imports was to leverage the Coq interpreter to see what variables and functions were available in different modules. An example of this is `UsualDecidableTypeFull`.

`UsualDecidableTypeFull` is a module type defined in the file *Structures/Equalities.v*. Its definition in the standard library is[1]:

```
Module Type UsualDecidableTypeFull <: DecidableTypeFull
 := UsualEq <+ UsualIsEq <+ UsualIsEqOrig <+ HasEqDec <+ HasEqBool.
```

which, without explaining the syntax in detail, essentially defines `UsualDecidableTypeFull` as a combination of other module types. Just looking at this definition doesn't tell us much about what `UsualDecidableTypeFull` actually contains. To see that, we can run the following code (from [3]):

```
Require Import Equalities.
Print UsualDecidableTypeFull.
```

which prints:

```
Module Type
 UsualDecidableTypeFull :=
 Sig
   Parameter t : Type.
   Definition eq : t → t → Prop.
   Definition eq_equiv : Equivalence eq.
   Definition eq_refl : forall x : t, x = x.
   Definition eq_sym :
     forall x y : t, x = y → y = x.
   ...
 End
```

[1] see the url https://github.com/coq/coq/blob/a4043608f704f026de7eb5167a109ca48e00c221/theories/Structures/Equalities.v#L261-L262.

Looking at this output (which is quoted in its entirety in [3]), we can identify, for example, that we have a proposition `eq_refl`, which is `forall x: t, x = x`, where `t` is specified by the module implementing this module type. In our case, `t` will be `nat`, so `eq_refl` will just be the proposition stating that any natural number is equal to itself.

By using this approach, I was able to more quickly identify relevant theorems in the standard library relating to `nat`.

By studying the standard library, I arrived at the following conclusions:

- The `nat` datatype and core functions are defined in files in the *Init* folder. These files also contain code relating to other types and functions operating on them.
- Many of the theorems relating to `nat` are in the *Arith* folder. Specifically, most are found in the file *Arith/PeanoNat.v*. However, it seems that the organisation of the code has changed over time, and many of the files in the *Arith* folder contain comments describing them as being obsolete, although they still contain some theorems.
- The *Arith/PeanoNat.v* file contains a module called `Nat` which implements module types specified in other files, such as *Numbers/Natural/Abstract/NAxioms.v* and *Structure/Equalities.v*. A lot of the theorems relating to natural numbers are in this one file, but this file imports many other files, which makes it confusing to understand.
- In general, the code relating to natural numbers is scattered around various files, some of which appear to be in the process of being removed.

3.2 Rewriting a subset of the standard library

In my rewrite of part of the standard library, I first started by defining the `nat` inductive type and some functions to operate on it:

```
Section nat_inductive_type.
  (* The nat inductive type *)

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

End nat_inductive_type.

Section functions.
  (* Some functions on nat *)

  Definition zero := O.
  Definition one := S zero.
```

```
Definition two := S one.

Definition succ := S.

Definition pred n :=
  match n with
  | O ⇒ O
  | S n' ⇒ n'
  end.

Fixpoint add n m :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (add n' m)
  end.

Definition double n := add n n.

Fixpoint mul n m :=
  match n with
  | O ⇒ O
  | S n' ⇒ add m (mul n' m)
  end.

Fixpoint sub n m {struct n} :=
  match n, m with
  | S n', S m' ⇒ sub n' m'
  | _, _ ⇒ n
  end.

Fixpoint eqb n m :=
  match n, m with
  | O, O ⇒ true
  | O, _ ⇒ false
  | _, O ⇒ false
  | S n', S m' ⇒ eqb n' m'
  end.

Fixpoint leb n m :=
  match n, m with
  | O, O ⇒ true
  | O, _ ⇒ true
  | _, O ⇒ false
  | S n', S m' ⇒ leb n' m'
  end.

Definition ltb n m := leb (S n) m.

Fixpoint max n m :=
  match n, m with
```

```
    | _, O ⇒ n
    | O, _ ⇒ m
    | S n', S m' ⇒ S (max n' m')
    end.

  Fixpoint min n m :=
    match n, m with
    | _, O ⇒ O
    | O, _ ⇒ O
    | S n', S m' ⇒ S (min n' m')
    end.

End functions.
```

I then defined some shorthand notation to make the definitions of theorems look more readable:

```
(* Some notation that allows us to avoid having to constantly
     have to write out some common function names.

   Note that section 2.4.2 remark 1 of the Coq reference
   manual says that the ‘Notation‘ command get cancelled when
   a section is closed. Thus we cannot put these notation
   commands into their own section without that section
   having to contain all other sections. *)

Notation "0" := zero.
Notation "x <> y" := (¬(eq x y)).
Notation "x = y" := (eq x y).
Notation "x + y" := (add x y).
Notation "x − y" := (sub x y).
Notation "x ∗ y" := (mul x y).
Notation "x < y" := (lt x y).
Notation "x <= y" := (le x y).
```

Then, out of the theorems found in the previous section, I selected several of them, and wrote proofs for them. I grouped these proofs into sections based on what functions they were testing properties of.

First, we have some theorems related to induction on natural numbers and the successor and predecessor functions:

```
The remainder of this file consists of theorems and proofs on
the functions and types defined above.
 *)
```

```
Section induction.
  (* These theorems concern induction over natural numbers. *)

  Theorem nat_case :
    forall (n:nat) (P:nat → Prop), P 0 → (forall m:nat, P (S m)) → P n.
  Proof.
    intros n P P_0. induction n.
    - intros _. exact P_0.
    - intro H. exact (H n).
  Qed.

  Theorem nat_double_ind :
    forall R:nat → nat → Prop,
      (forall n:nat, R 0 n) →
      (forall n:nat, R (S n) 0) →
      (forall n m:nat, R n m → R (S n) (S m)) → forall n m:nat, R n m.
  Proof.
    intros R H0 H1 H2 n. induction n as [| n' IHn].
    - exact H0.
    - destruct m.
      + exact (H1 n').
      + apply H2. exact (IHn m).
  Qed.

End induction.

Section succ_and_pred.
  (* These lemmas involve successor and predecessor functions. *)

  Lemma one_succ: one = succ zero.
  Proof. reflexivity. Qed.

  Lemma two_succ: two = succ one.
  Proof. reflexivity. Qed.

  Lemma pred_succ: forall n : nat, (pred (succ n)) = n.
  Proof. reflexivity. Qed.
End succ_and_pred.
```

Next, we have some theorems related to use of <>, which allows constructing propositions asserting that two terms are not equal:

```
Section not_equal.

  Theorem O_S : forall n:nat, 0 <> S n.
  Proof. discriminate. Qed.

  Lemma S_injective: forall x y: nat, S x = S y → x = y.
```

```
 (*
Initially, I tried to prove this lemma by doing induction on x and y
and dealing with each base case and induction case separately, but
I couldn't complete a proof in this manner.

induction x.
  - induction y.
    + reflexivity.
    + intros H. discriminate H.
  - induction y.
    + intros H. discriminate H.
    + intros H. Admitted.

Then I read chapter 6 of the book Coq'Art and read about the
injection tactic. The following is a valid proof of this
lemma and uses that tactic:

intros x y H.
injection H. trivial.

Below I have done a more detailed proof that gives a bit
more detail as to how the injection tactic works.
  *)
 Proof.
   intros x y H.
   change (let f (n: nat) :=
               match n with
               | O ⇒ x
               | S n' ⇒ n'
               end
           in f (S x) = f (S y)).
   rewrite H. reflexivity.
 Qed.

 Theorem n_Sn : forall n:nat, n <> S n.
 Proof.
   intros n. elim n.
   - discriminate.
   - intros n0 H H1.
     apply H. apply S_injective.
     rewrite ← H1. reflexivity.
 Qed.

End not_equal.
```

The next section concerns addition. First, we start by proving some simple properties of addition, such as addition of 0 to any number leaving that number unchanged. Then, we construct proofs of successively more complicated theorems,

eventually culminating in proving that addition is commutative and associative (we also prove a theorem called `plus_swap` which is used later for proving theorems relating to multiplication). The ordering of some of the proofs here was inspired by [4].

```
Section addition.

  Lemma plus_O_n : forall n:nat, 0 + n = n.
  Proof. reflexivity. Qed.

  Theorem plus_n_O : forall n:nat, n = n + 0.
  Proof.
    intros n. induction n as [| n' IHn].
    - reflexivity.
    - simpl. rewrite ← IHn. reflexivity.
  Qed.

  Lemma plus_Sn_m : forall n m:nat, S n + m = S (n + m).
  Proof. reflexivity. Qed.

  Theorem plus_n_Sm : forall n m : nat, S (n + m) = n + (S m).
  Proof.
    intros n m. induction n as [| n' Ihn'].
    - reflexivity.
    - simpl. rewrite → Ihn'. reflexivity.
  Qed.

  Theorem plus_comm : forall n m : nat, n + m = m + n.
  Proof.
    intros n m. induction n as [| n' Ihn'].
    - simpl. rewrite ← plus_n_O. reflexivity.
    - simpl. rewrite → Ihn'. rewrite → plus_n_Sm. reflexivity.
  Qed.

  Theorem plus_assoc : forall n m p : nat, n + (m + p) = (n + m) + p.
  Proof.
    intros n m p. induction n as [| n' Ihn'].
    - reflexivity.
    - simpl. rewrite → Ihn'. reflexivity.
  Qed.

  Theorem plus_swap : forall n m p : nat, n + (m + p) = m + (n + p).
  Proof.
    intros n m p.
    assert (H: n + (m + p) = n + m + p).
    {rewrite → plus_assoc. reflexivity. }
    assert (H1: n + m = m + n).
    {rewrite → plus_comm. reflexivity. }
    rewrite → H. rewrite → H1. rewrite → plus_assoc. reflexivity.
  Qed.
```

```
End addition.
```

We also prove two theorems related to subtraction:

```
Section subtraction.

  (* This lemma looks simple, but as the structure of n can't be
     determined,
     we need to do a proof by induction *)
  Lemma sub_0_r : forall n : nat, n − zero = n.
  Proof.
    intros n. elim n.
    − reflexivity.
    − reflexivity.
  Qed.

  Lemma sub_succ_r : forall n m : nat, n − (succ m) = pred (n − m).
  Proof.
    induction n as [| n' IHn].
    − intros m. reflexivity.
    − intros m. elim m.
      + simpl. apply sub_0_r.
      + intros n0 H. simpl. apply IHn.
  Qed.
End subtraction.
```

Finally, we prove some theorems related to multiplication. Like the section on addition, we start with some simple theorems, and successively prove more complex theorem. We culminate with a proof that multiplication is commutative:

```
Section multiplication.

  Theorem mult_0_l: forall n: nat, 0 * n = 0.
  Proof. reflexivity. Qed.

  Theorem mult_0_r: forall n:nat, n * 0 = 0.
  Proof.
    intros n. induction n as [| n' Ihn'].
    − reflexivity.
    − simpl. rewrite → Ihn'. reflexivity.
  Qed.

  Lemma mult_n_Sm : forall n m:nat, n * m + n = n * S m.
  Proof.
    intros n m. induction n as [| n' IHn].
```

```
    - reflexivity.
    - simpl.
      rewrite ← IHn. rewrite ← plus_n_Sm. rewrite plus_assoc.
      reflexivity.
  Qed.

  Lemma mult_succ_l: forall n m: nat, (S n) * m = (n * m) + m.
  Proof.
    intros n m. induction n.
    - simpl. rewrite ← plus_n_O. reflexivity.
    - simpl. rewrite plus_comm. reflexivity.
  Qed.

  Lemma mult_plus_S: forall m n : nat, n + (n * m) = n * (S m).
  Proof.
    intros n m.
    induction m as [| m' Ihm'].
    - reflexivity.
    - simpl. rewrite ← Ihm'. rewrite → plus_swap. reflexivity.
  Qed.

  Theorem mult_comm : forall m n : nat, m * n = n * m.
  Proof.
    intros n m. induction n as [| n' Ihn'].
    - rewrite → mult_0_r. reflexivity.
    - simpl. rewrite → Ihn'. rewrite ← mult_plus_S. reflexivity.
  Qed.

End multiplication.
```

Chapter 4

Conclusion

In this report, I set out to understand how to use Coq, and then to use that understanding to rewrite some part of the standard library. Understanding how to use Coq took longer than expected, and proving theorems from the standard library also involved many difficulties, as it often isn't clear what the best way is of trying to prove a theorem. Also, unlike proving a theorem by hand, the Coq interpreter forces the user to be unambiguous when writing proofs, and will reject any proofs that are incorrect (and so do not type check). One must also write proofs in a way that can be understood by the Coq interpreter, which is not often straightforward.

Working on this project introduced me to the power of dependently typed functions, and what possibilities they open up for more rigorous type checking. I intend to look further into this topic in the near future.

Coq is a very powerful tool. This report covers only the basics of what can be done with Coq. In particular, this report did not make mention of certified programs, where functions that perform computations can return *certificates*, or proof terms that express the correctness of the function.

I would also like to thank my supervisor, Dr Nicola Gambino, for his invaluable guidance throughout this project, as well as helping me to understand some of the more advanced concepts covered in this report.

Bibliography

1. Bertot Y, Castéran P. *Interactive Theorem Proving and Program Development.* Springer-Verlag Berlin Heidelberg, 2004.

2. The Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA, available at https://coq.inria.fr/refman/, 2016.

3. Suleman I. *Coq Standard Library Map.* available at https://finalyearuniproject.ismail-s.com/projectmap.html, 2017.

4. Pierce BC, Amorim AA, Casinghino C *et al. Software Foundations.* Self-published, available at https://www.cis.upenn.edu/~bcpierce/sf/current/index.html, 2017.