Chapter 1

Reading_HoTT_in_Coq

1.1 Reading Coq Files

If you're reading a PDF file, it was generated from a Coq file using Coq's documentation tool, "coqdoc".

If the file extension is ".v", you have the original Coq file and can read it in any text editor ("Notepad" on Windows, "TextEdit" on Macs). If you've installed CoqIDE, Coq's graphical editor, you can open it there and verify the proofs. It can also be used on the web at http://prover.cs.ru.nl/

This document is a light-weight introduction to Coq using examples from the first half of the book "Homotopy Type Theory". Our goal is to give readers a taste of using a proof assistant. When you're done, you will be able to read what other people have proven in Coq and possibly prove some simple theorems of your own.

If you are interested in going further with Coq, at the end of this document are instructions on how to install Coq, links to full tutorials, and links to the Coq Reference Manual.

This file is written in "plain" Coq 8.4. This so it can be used with the official distribution or on the web at http://prover.cs.ru.nl/. This will prevent us from doing proofs that use "higher inductive types", a feature that is only available with a special version of Coq.

While this file stands on its own, where possible, we use the same theorem names as the "HoTT" library for Coq. When you're done reading this file, you should be able to read most of what has been proven in that library.

1.1.1 Background

Coq is a good platform for homotopy type theory ("HoTT") work. Coq has a long history - created in 1984 - and is supported by INRIA. Coq has a number of features that have allowed it to be used for significant proofs. Those proofs include the formalization of the Four Color Theorem and the Feit-Thompson Theorem.

Coq uses a dependant type theory derived from “The Calculus of Constructions”. It differs from Martin-Loef’s intentional type theory, but, as we’ll see, its propositional equality has the same higher-groupoid structure that allows us to do HoTT.

1.2 Introduction

Coq mostly works with two concepts:

- dependant functions (Π-types)
- inductive types

Inductive types are used to implement the common types of type theory: dependent pairs (Σ-types), disjoint unions, etc.. We’ll use those types as many of our examples.

We’ll start with the familiar example of Peano’s natural numbers.

1.2.1 Natural Numbers

The book’s description of natural numbers says:

- the type N:U of natural numbers.

whose elements are constructed using

- 0:N, and
- the successor operation succ:N->N.

The equivalent in Coq is:

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

As you can see, Coq’s default library uses different names:

- *nat* instead of “N”,
- `Set` instead of “U”,
- *O* (the capital letter “oh”) instead of “0”, and
- *S* instead of “succ”.

The command `Inductive` creates a new type. In this case, the type is called *nat*. The type *nat* will live in the universe type called `Set`, which in Coq is the first universe or "the universe of small types".

After the := symbol comes the constructors for the new type. The first says O (the capital letter "oh") is a term of type *nat*. The second says S is a function from *nat*s to *nat*s.

After the last constructor is a period ("."). The command that started with the word "Inductive" ends at the period. Every command in Coq ends with a period.

Properties of Constructors

The constructors of an inductive type have a number of properties. In a rough description, the major properties are:

- Constructors are axiomatic. The function S exists without any definition - while it can be called, it cannot be evaluated.
- The constructors are exhaustive. There is no other way to create a term of type *nat*.
- Contructors create well-founded terms. There is no way to create a self-referential *nat*, such as having an S that returns itself.
- Terms created with different constructors are not equal. O is not equal to S called with any other *nat*.

If you know Peano's Axioms, these should all seem familiar. But these conditions apply to every inductive type, not just *nat*.

NOTE: Homotopy type theory changes some of these properties. The univalence axiom and the path constructors of higher inductive types create new elements of the identity type. Higher inductive types can define terms created with different constructors as propositionally equal.

Examples of Natural Numbers

Coq's command `Check` will print out the type of a term. Obviously, O is a valid term and the `Check` will print the type *nat*. (Remember, O here is the capital letter "oh".)

```
Check O.
```

To get the number one, we have to call the function S. A function call (or "function application") in Coq is done by juxtaposition, so

- gcd 4 6

instead of

- gcd(4, 6)

Thus, the number one is written:

```
Check S O.
```

For the number two, we have to add parentheses so that the second S is interpreted as a function call rather than as a second argument to the first S.

```
Check S (S O).
```

The number five is

```
Check S (S (S (S (S O)))).
```

By default, Coq loads a plugin that interprets decimal numbers as *nat*s.

```
Require Import Datatypes.
Declare ML Module "nat_syntax_plugin".
```

Thus, we could have checked the number five simply by doing

```
Check 5.
```

Now that we have seen the basics of Coq's inductive types, let's see its other main feature: dependent functions.

1.2.2 Identity function

We'll start by defining the identity function on natural numbers and then we'll write a dependantly-typed identity function that works for any type.

The identity function (or identity map) for natural numbers is:

```
Definition idmap_nat : nat -> nat :=
  fun (n:nat) => n.
```

The `Definition` command assigns a value of a given type to a name. Its format is:

- Definition <name> : <type> := <value> .

In our example, the name is "idmap_nat". The type is a function from *nat* to *nat*. (Notice how the "->" operator approximates the arrow used in the book for non-dependantly typed functions.) The value for "idmap_nat" is a function.

In Coq, a function is written:

- fun <params> => <term>

In the book, this would have been written as

- <param> $\mapsto$ <term>

or

- λ <param>.<term>

In the example, the function has one parameter, "n", which has type *nat*. The function's result is simply "n" itself, since this is the identity function.

Shorthand

Coq has a shorthand for defining functions. Here is the identity function for *nat*s again.

```
Definition idmap_nat_short (n:nat) : nat :=
  n.
```

Notice that the parameter is put immediately after the function's name and we no longer need "fun ... =>".

Examples of idmap_nat

The Coq command `Compute` will evaluate a function and print the result.

```
Compute idmap_nat (S O).
```

prints (S O), like an identity function should.

Dependant types

"idmap_nat" is not dependantly typed, so we were able to use the arrow ("->") to denote its type. We could have written the function's type as if it was dependantly typed. In the book, dependant types are declared with a capital "Π". Coq uses the keyword `forall`.

```
Definition idmap_nat_dep : forall nn:nat, nat :=
  fun n:nat => n.
```

In Coq, a dependent function type is written:

- forall <params> , <type>

In the book, this would have used capital Π and a subscript:

- Π (<param>) <type>

As you'd expect, the names in the param list of a `forall` expression are only bound inside the "<type>" part of the `forall`. In this example, the parameter "nn" cannot be used when defining the function. (Usually, we use the same name in the `forall` and the `fun` parts; different ones were used here to demonstrate the point.)

A `forall` can have multiple parameters. If a parameter is dependently typed on another parameter, the dependent one must come later in the list. (We'll see an example soon.)

Now that we know how to write a dependant function type, we can write an identity function that works for any type.

```
Definition idmap : forall A:Type, A -> A :=
  fun (A:Type) (x:A) => x.
```

"idmap" is a dependantly-typed function: its return type depends on the type of its first parameter. Therefore, we had to use the `forall` operator for that parameter.

In Coq, the type "Type" refers to *some* universe type. Coq will do the work of figuring out which universe type, as long as we don't implicitly cause impredicativity.

We can, of course, rewrite the definition of "idmap" using Coq's shorthand for functions.

```
Definition idmap_short (A:Type) (x:A) : A :=
  x.
```

Examples of idmap

```
Compute idmap nat (S (S O)).
```

Prints (S (S O)), like an identity function should.

Coq has a number of features for making it easier to define and call functions. We've already seen the "shorthand" for definitions. In the rest of this section, we'll see:

- partial application,
- implicit arguments, and
- type inferencing.

Partial application

Since we've defined the identity function for any type, we can now define the identity function for *nat*s in terms of it.

```
Definition idmap_nat_from_idmap : nat -> nat :=
  idmap nat.
```

The value "idmap nat" is a function call. It calls the function "idmap" with the type *nat*. Since "idmap" expected 2 arguments and we only provided 1, this is called a "partial application". The result of the partial application is a function that is still waiting for 1 more argument. That function is the identity function on *nat*s and this command assigns it a name.

We can check this new function by passing the second of the two arguments.

```
Compute idmap_nat_from_idmap (S (S (S O))).
```

Implicit Arguments

Calling "idmap nat (S O)" seems repetitive because Coq can determine that "(S O)" has type *nat*. (Remember, in type theory, an element can belong to only one type.) We can use Coq's implicit arguments feature to tell Coq to always infer some argument values.

Curly braces are used to mark a parameter for implicit arguments.

```
Definition idmap_implicit {A:Type} (x:A) : A :=
  x.
```

Now, we can call the general identity function with just one argument.

```
Compute idmap_implicit (S O).
```

Implicit arguments are usually handy, but sometimes they get in the way. Before, we declared a version of "idmap_nat" by calling "idmap". We did it by passing "nat" as the first parameter to "idmap".

If we call "idmap_implicit" with "nat", Coq will assume that *nat* is "x" and use the type of *nat*, which is `Set`, for the implicit parameter. Obviously, we don't want that. We can prevent Coq from using implicit arguments by putting an "at sign" ("@") in front of the function name.

```
Definition idmap_nat_from_idmap_implicit : nat -> nat :=
  @idmap_implicit nat.
Compute idmap_nat_from_idmap_implicit (S O).
```

Another way to mark parameters for implicit arguments is with the "Arguments" command. Because that command has a lot of features and complex syntax, we won't go into its details in this document. Nonetheless, we will use the command so that our examples look like those of the HoTT Coq library.

Type Inferencing

In many cases, Coq can infer the type of a parameter or even a whole function without us have to state it explicitly.

```
Definition idmap_inferred {A} (n:A) :=
  n.
```

Here, both the type of "A" (which is `Type`) and the returned type of the function (which is "A") are both inferred. As you can see, this allows very concise definitions.

```
Compute idmap_inferred (S (S O)).
```

1.2.3 Addition of Natural Numbers

As the last part of our introduction, we define addition as a function on natural numbers and show how to use the operator "+" to call it.

Induction

When we issued the command `Inductive` to create the type *nat*, Coq also created a function "nat_rect" for induction on natural numbers. Its type is:

```
  nat_rect
     : forall P : nat -> Type,
        P 0 ->
        (forall n : nat, P n -> P (S n)) ->
        forall n : nat, P n
```

This is identical to the induction constant named “ind_N” in the HoTT book. We can use this function to define addition.

```
Definition plus (m n: nat) : nat :=
  nat_rect (fun _ => nat) n (fun m' sum => S sum) m.
```

The function “plus” is defined by a call to “nat_rect” with 4 arguments:

The first argument determines the type of the result. Addition always results in a *nat*, so the first argument is a function that always returns the type *nat*. When specifying the function, we used underscore ("_") which is a special parameter name that indicate to Coq that the parameter isn’t used in the function. (Multiple parameters can be named "_" if they are all not needed.)

The second argument to “nat_rect” is “n”. This is the base case; the result when “m” is zero.

The third argument is the inductive case. It takes “m”-prime and the result (sum) upto “m”-prime and produces the result for the successor of “m”-prime, which is just the sum plus one.

The fourth argument is “m”, the value to calculate the sum at.

```
Compute plus 4 2.
```

In this example, we directly called the induction constant “nat_rect”. This is one way to do induction in Coq. The other way is similar to the “pattern matching” describing in the HoTT book.

Match Expressions

Addition can also be defined using a “match” expression.

```
Fixpoint plus_using_match (m n: nat) : nat :=
  match m with
      | 0 => n
      | S m' => S (plus_using_match m' n)
  end.
```

The `match` expression represents case analysis on an element of an inductive type. Since every canonial element of an inductive type must have been made with a constructor, the `match` expression gives a value that depends on which constructor was used. The `match` expression is very expressive, but a simplified understanding is:

```
  match <element> with
    | <constructor_pattern> => <value>
    | <constructor_pattern> => <value>
     ...
  end
```

NOTE: In HoTT, not every element of the identity type is made with a constructor. Nonetheless, this case analysis still works. See the HoTT book for an explanation.

In our example, "m" is treated as a canonical *nat*. If "m" was made with the constructor constant O (capital-oh), the value of the `match` expression is "n". If "m" was made with the constructor function S called with some other *nat*, called "m"-prime here, then the value of the match expression is the successor of "m"-prime plus "n".

Notice that the function is defined in terms of itself. In order to allow that, we had to use command `Fixpoint` instead of the usual `Definition`.

Coq will transform the match expression into a call to "nat_rect". If it cannot, Coq will print an error message.

Notations

We could always represent addition with "plus 4 2", but it is more natural to read and write "4 + 2". We tell Coq to use this format through `Notation` command.

```
Notation "n + m" := (plus n m) : nat_scope.
Open Scope nat_scope.
```

Now we can write.

```
Compute 4 + 2.
```

We've actually already been using a notation. The "->" operator is defined as:

```
Reserved Notation "x -> y" (at level 99, right associativity, y at level 200).
Notation "A -> B" := (forall (_ : A), B) : type_scope.
```

Because `Notations` could conflict, every `Notation` goes into a scope. Here, the scopes are called "nat_scope" and "type_scope". When a scope is opened, all of its notations become available to be used. If two notations in open scopes conflict, the one opened more recently is used. Here, "nat_scope" was opened by the command "Open Scope nat_scope". The other, "type_scope", is special and is open anywhere a type is expected.

If we ever wanted to stop using the plus `Notation`, we could issue the command "Close Scope nat_scope".

A more complex example is:

```
Definition compose {A B C : Type} (g : B -> C) (f : A -> B) :=
  fun x => g (f x).
```

```
Notation "g 'o' f" := (compose g f) (at level 40, left associativity).
```

The single quotes are used to turn the letter "o" (small-oh) into an operator.

"at level 40" indicates the precedence of the operator. A lower precedence level means that an operator "binds more tightly". That is, that a `Notation` is selected over another. Thus, for natural numbers, multiplication is at level 40, while addition is at 50. (Those operators have default precedences set by a "Reserved Notation" command.)

"left associativity" is what you would expect.

That covers the basics of reading Coq theorems. The rest of this document goes over the types, functions, and operators that are commonly used in type theory and in HoTT.

1.3 Common Types of Type Theory

After a short discussion about universe types in Coq, we go through how each of the types that are commonly used in type theory are implemented as inductive types.

1.3.1 Universes

The HoTT book describes an infinite hierarchy of universes U_0, U_1, U_2, ... that are cumulative. That is, that every type in a universe is also in every higher universe.

Coq's universes have a similar structure, except that the lowest universe is split into two: "Prop" and "Set".

The "Prop" universe contains propositions - statements that can be proven or disproven. In practice, that means types that are shown to be either inhabited or uninhabited. Types in "Prop" must be "proof irrelavent": it cannot matter which term inhabits the type, just that it is inhabited.

The "Set" universe contains all other "small types". (Small types are ones that do not contain references to a universe.) Since in homotopy type theory every equality proof is relavent, all of our inductive types will reside in the "Set" universe.

The infinite number of universes above "Prop" and "Set" are known as "Type(1)", "Type(2)", "Type(3)", etc. However, the user only ever has to enter "Type". Coq will, behind the scenes, assigned a numbered universe to every usage of "Type", as long as there is no impedicativity. (If you implicitly cause impredicativity, you'll see an error message.)

1.3.2 Dependent Function Types

The most commonly used type in type theory is the function. This type in Coq was covered in detail earlier. A summary of its usage is:

The dependent type:

- Book: Π (<params>) <type>
- Coq: forall <params> , <type>

The non-dependent type:

- Book: <type> $\rightarrow$ <type>
- Coq: <type> -> <type>

An unnamed function:

- Book: <param> $\mapsto$ <term>
- Book: λ <param> . <term>

- Coq: fun <params> => <term>

A function call (or "function application"):

- Book: <fun>(<arg1>, <arg2>)
- Coq: <fun> <arg1> <arg2>

Function composition:

- Book: <fun> ∘ <fun>
- Coq: <fun> o <fun>

1.3.3 Non-Dependent Pair Types

In the HoTT book, the non-dependent pair type can be considered a special case of a dependent pair. For example, the projection functions work the same on both the non-dependent and dependent types. In Coq, however, it is more convenient to have two separate types.

In the book, the non-dependent pair type, or cartesian product, requires

- a type A : U, and
- a type B : U

and is written

- A × B

In Coq, the inductive type is written:

```
Inductive prod {A B:Type} : Type :=
  pair : A -> B -> @prod A B.
```

Here, "pair" is a constructor that takes two arguments, an element of type "A" and an element of type "B", and produces an element of type "prod A B". So, "prod A B" is the type of non-dependent pairs and "pair a b" creates a pair.

This inductive type definition uses a shortcut. The types "A" and "B" are used in the constructor "pair", but are not listed as parameters. This is because any parameters listed immediately after the type name ("prod") are treated as parameters to both the type of "prod" and to all constructors.

An equivalent (but longer) definition of "prod" would be

```
Inductive prod_long : Type -> Type -> Type :=
  pair_long : forall {A B: Type}, A -> B -> prod_long A B.
```

When Coq creates "prod", it also creates the induction function "prod_rect". This is the inductive constant, similar to "nat_rect" we mentioned earlier. What we didn't say earlier is that there are *two* other induction functions: "prod_rec" and "prod_ind" (as well as "nat_rec" and "nat_ind").

The reason for three induction constants is that there are three kinds of universes: "Prop", "Set", and "Type". The function "prod_rect" and "nat_rect" puts the types they create in the "Type" universe. Likewise, "prod_rec" and "nat_rec" put the resulting type in "Set" and "prod_ind" and "nat_ind" put it in "Prop". When doing HoTT, it is usually safe to just use the "rect" function.

When we use a `match` expression for induction, Coq uses type inferencing to chose the correct induction function.

Speaking of duplication, Coq has a second non-dependent pair type called "and". This type takes arguments from the "Prop" universe and puts the resulting type in "Prop". But, since we're not using "Prop", we won't cover it here.

Pair Notation

From the definition of "pair_long", it is clear that the constructor takes 4 parameters: two types and an element of each type. But given the two elements, Coq can always infer their types. Thus, we can create a pair using just the two elements. For example:

```
Check (pair 4 2).
```

In the book, we use "A $\times$ B" to denote the type and "(a,b)" to denote a pair. Coq allows us to use a similar syntax by using the `Notation` command.

```
Notation "x * y" := (@prod x y) : type_scope.
Notation "( x , y , .. , z )" := (pair .. (pair x y) .. z) : core_scope.
```

Now when Coq sees "nat * nat", it will translate it into "(prod nat nat)" and, likewise, translate "(4, 2)" into "(pair 4 2)". The second `Notation` command will also convert tuples of any length into pairs-within-pairs.

Now, we can write the type and elements of dependent pairs like we're accustomed.

```
Check (nat * nat)%type.
Check (4,2).
```

The "%type" forces Coq to use the "type_scope" to interpret the expression. It is needed because `Check` does not expecting a type.

Projection functions

The projection functions extract the first or second part of a pair. For non-dependent pairs in Coq, these are called "fst" and "snd".

```
Section projections.
  Context {A : Type} {B : Type}.
  Definition fst (p:A * B) :=
    match p with
      | (x, y) => x
    end.
  Definition snd (p:A * B) :=
    match p with
      | (x, y) => y
    end.
End projections.
```

The section feature is used here to simplify the list of parameters. The section starts with "Section projections" and ends at "End projections". The statement "Context ..." signals that "A" and "B" are parameters to every subsequent definition that uses them inside the section. Thus, both "fst" and "snd" have two type parameters and, because curly braces ("{", "}") were used, those parameters are implicit.

In the `match` expressions, the constructor function "pair" is written using the notation "(x, y)".

```
Compute fst (4,2).
Compute snd (4,2).
```

1.3.4 Dependent Pair Types

In the HoTT book, the dependent pair type, also called Σ-type, requires

- a type A : U, and
- a type family P : A -> U

and is written

- Σ (x:A) P(x)

In Coq, it is defined by:

```
Inductive sigT {A:Type} (P:A -> Type) : Type :=
    existT : forall x:A, P x -> sigT P.
```

Here, "existT" is the constructor that takes two arguments, an element "x" of type "A" and an element (unnamed) of type "P x", and produces an element of "sigT A P". So, "sigT A P" is the type of pairs and "existT x p" creates a pair (when "p" has type "P x").

If you read closely, you'll see that the type produced by "existT" is "sigT P" not "sigT A P". The "A" can always be inferred from "P".

To repeat (for the last time), every time Coq creates a new inductive type, like "sigT" here, Coq also creates three induction functions, "sigT_ind", "sigT_rec" and "sigT_rect". These functions put their results into the "Prop", "Set", and "Type" universes (respectively).

Most of the time, we don't care, since we'll use a `match` expression that infers which induction function to use.

Like "prod" with "and", Coq has dependent pair types besides "sigT". The types "ex" (short for "there exists") and "sig" both act as dependent pairs, but use the "Prop" universe. Again, we're not using "Prop", we won't cover them here.

Pair Notation

To create a dependent pair, we must supply a function taking an element of the first type to a type for the second element. Since we haven't defined propositional equality, we can't do much that is interesting here. For now, we can create a pair of *nat*s by supplying a function that always returns *nat*.

```
Check (existT (fun _:nat => nat) 4 2).
```

Obviously, that expression is long to write and difficult to read and we want to use a `Notation` for it. Since Coq already uses "(a,b)" for non-dependent pairs, the HoTT Coq library uses the semicolon here.

```
Notation "{ x : A & P }" := (sigT (fun x:A => P)) : type_scope.
Notation "( x ; y )" := (existT _ x y) : fibration_scope.
Open Scope fibration_scope.
```

When Coq sees "(4;2)", it will translate that into "(existT _4 2)". The underscore ("_") in a function application indicates that Coq should try to infer the argument or ask for help from the user.

Here the dependent-pair `Notation` goes into the "fibration_scope". Since that is a new scope, we must "Open" it to make the `Notation` available.

Below is an example using the dependent pair "(4;2)". It is necessary to say what its type is, so that Coq can infer the hidden argument to "existT".

```
Definition dep_pair_example_type :=
  { x:nat & nat }.
Definition dep_pair_example : dep_pair_example_type :=
  (4;2).
Check dep_pair_example.
```

Projection functions

The projection functions extract the first or second part of a pair. For dependent pairs in Coq, these are called "projT1" and "projT2".

```
Section Projections.
  Context {A : Type}.
  Context {P : A -> Type}.
  Definition projT1 (x:sigT P) : A :=
    match x with
```

```
      | existT a _ => a
    end.

  Definition projT2 (x:sigT P) : P (projT1 x) :=
    match x return P (projT1 x) with
      | existT _ h => h
    end.

End Projections.
```

These are pretty much as you'd expect. There are two items worth commenting on.

The underscore ("_") in the constructor pattern is used to indicate an unused parameter in the `match` expression. We've seen this before in parameters to `fun` and `forall`.

The other feature worth commenting on is the "return <type>" in the `match` expression of "projT2". This syntax is used when the `match` expression has a type that depends on the element of the inductive type being matched on.

We will not go into all the details on the variations of the `match` expression, because this document is about reading what has been proven - that is, the *type* of an expression - and not about understanding the proof - which is the value of the expression.

The `Notations` for the projectors are:

```
Notation "x .1" := (projT1 x) (at level 3) : fibration_scope.
Notation "x .2" := (projT2 x) (at level 3) : fibration_scope.
```

And some examples of it are:

```
Check (dep_pair_example .1).
Check (dep_pair_example .2).
```

1.3.5 Disjoint Union Type

In the HoTT book, the disjoint union type, also called coproduct, requires

- a type A : U, and
- a type B : U

and is written

- A + B.

In Coq, it is defined by:

```
Inductive sum (A B:Type) : Type :=
  | inl : A -> sum A B
  | inr : B -> sum A B.

Arguments inl {A B} _ , [A] B _.
Arguments inr {A B} _ , A [B] _.
```

```
Notation "x + y" := (sum x y) : type_scope.
```

Ignoring the "Arguments" command, which we aren't covering in this document, the rest should be familiar by now.

Since the type "sum" has two constructors, "inl" and "inr", we can have two examples that build an element of a type.

```
Definition dijoint_union_example_type :=
  (nat + (nat * nat))%type.
Definition dijoint_union_example1 : dijoint_union_example_type :=
  inl 4.
Definition dijoint_union_example2 : dijoint_union_example_type :=
  inr (4,2).
```

Likewise, any `match` expression needs to handle both constructors.

```
Definition left_or_first (a : dijoint_union_example_type) : nat :=
  match a with
      | inl x => x
      | inr p => fst p
  end.
```

1.3.6 Zero, One, and Two Types

The finite types with 0, 1, and 2 elements play special roles in type theory. In standard Coq those types are:

```
Inductive Empty_set : Set :=.
```

```
Inductive unit : Set :=
    tt : unit.
```

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

The HoTT Coq library uses slightly different names for the types. (Although the constructors have the same names.)

```
Definition Empty := Empty_set.
Definition Unit := unit.
Definition Bool := bool.
```

Standard Coq also has finite types that live in the "Prop" universe. The type "True" has one constructor and the type "False" has zero.

Not operator

In HoTT, the not operator indicates that elements of a type can be mapped to the elements of the empty (zero) type.

```
Definition not (A:Type) : Type := A -> Empty.
Notation "~ x" := (not x) : type_scope.
```

In Standard Coq, logic is usually done in the "Prop" universe, so this operator maps to the type "False" that lives there (instead of "Empty" which lives in "Set").

Absurdity Implies Anything

Obviously, a `match` expression for the *Unit* type handles one constructor and the match expression for the *Bool* type handles two constructors. But what about the *Empty* type? It has no constructors, so its `match` expression is empty. In logic, this is the equivalent of "ex falso quodlibet" or "from contradiction, anything".

```
Definition contradiction_implies_anything (a:Empty) (C:Type) : C :=
  match a with
      end.
```

The induction constant for Empty is

```
  Empty_rect : forall (P : Empty -> Type) (e : Empty), P e
```

1.3.7 Identity Type

The identity type is defined as:

```
Inductive paths {A : Type} (a : A) : A -> Type :=
  idpath : paths a a.
```

Where "paths" equates to "Id" in the HoTT book and "idpath" to "refl".

The standard Coq library defines equality using a type "eq" with constructor "refl". This type is different from "paths" because "eq" is in the "Prop" universe and its elements are *not* proof-relevant. To do homotopy type theory, we need an equality that is proof-relevant and exists in the "Type" universe.

The operator for the identity type is the equal sign. There is also a `Notation` that allows the user to explicitly state the type.

```
Notation "x = y :> A" := (@paths A x y) : type_scope.
Notation "x = y" := (x = y :>_) : type_scope.

Arguments idpath {A a} , [A] a.
Arguments paths_ind [A] a P f y p.
Arguments paths_rec [A] a P f y p.
Arguments paths_rect [A] a P f y p.

Notation "1" := idpath : path_scope.
Local Open Scope path_scope.
```

1.4 Homotopy Type Theory

Now that we have seen the common types of type theory, we can use them to do homotopy type theory. This section will demonstrate some theorems of HoTT and introduce the types used to do HoTT in Coq. Because this file is using "standard" Coq, we cannot demonstrate higher inductive types.

1.4.1 Properties of Paths

For every element of a type, there is a constant path. This is the same notion as equality being reflexive. This property is witnessed by "idpath", which has type:

```
@idpath
    : forall {A : Type} (a : A), a = a
```

The "theorem" of reflexivity can be stated "for every type and for every element of that type, there is an equality with that element equal to itself". In Coq, that theorem is "proven" by function that takes a type, an element of that type, and returns an element witnessing the equality.

For example, if we wanted to demonstrate that "4=4", we could do:

```
Check @idpath nat 4.
```

which is an element that has type "4=4". Obviously, with implicit arguments, we do not need "nat" and can use just "idpath 4". Not so obviously, we can go a step further. If type inferencing can determing the type returned by "idpath", such as "4=4" in our example, then implicit arguments can fill in the "4" as well! So most of the time you will just see "idpath" or its `Notation`, "1" (in the "path_scope" scope).

```
Check idpath : 4 = 4.
Check 1 : 4 = 4.
```

Next, we prove that every path has an inverse. (Or, "equality is symmetric".) In Coq, this proof looks like a function that takes any path and returns its inverse.

```
Definition inverse {A : Type} {x y : A} (p : x = y) : y = x
  := match p with
       | idpath => idpath
     end.
```

```
Arguments inverse {A x y} p : simpl nomatch.
```

```
Notation "p ^" := (inverse p) (at level 3) : path_scope.
```

This `match` expression hides a number of type inferences and implicit arguments. The type of "p" is "paths A x y", which had to be constructed using "idpath A x" with "y" being the same as "x". The value returned by the match has type "paths A y x", so Coq can infer that the arguments to "idpath" are "A" and "x" and, because "y" is the same as "x", intrepret the resulting "A x x" as "A y x".

Notice how this proof is similar to the HoTT book's proof where "y" is assumed to be the same as "x" and "refl_x" is mapped to "refl_x".

Next, we prove that paths concatenate. (Equality is transitive.) Like before, this is a function that takes any path from "x" to "y" and any path from "y" to "z" and returns a path from "x" to "z".

```
Definition concat {A : Type} {x y z : A} (p : x = y) (q : y = z) : x = z :=
  match p, q with
     | idpath, idpath => idpath
  end.
```

```
Arguments concat {A x y z} p q : simpl nomatch.
```

```
Notation "p @ q" := (concat p q) (at level 20) : path_scope.
```

The comma in the `match` expression is part of the `match` syntax. It is *not* a non-dependent pair. It is a shortcut that allows two inductions to be done using a single `match` expression. When the `match` gets translated into two calls to "paths_rect", we don't care in which order the calls happen; the results are the same. As the HoTT book explains, this proof could be done with just one call to "path_rect", but the result from a single induction would not behave symmetrically.

The following proofs show the relationship of "idpath", "inverse" and "concat".

```
Definition concat_p1 {A : Type} {x y : A} (p : x = y) : p @ 1 = p :=
  match p with idpath => 1 end.
Definition concat_1p {A : Type} {x y : A} (p : x = y) : 1 @ p = p :=
  match p with idpath => 1 end.

Definition concat_pV {A : Type} {x y : A} (p : x = y) : p @ p^ = 1 :=
  match p with idpath => 1 end.
Definition concat_Vp {A : Type} {x y : A} (p : x = y) : p^ @ p = 1 :=
  match p with idpath => 1 end.

Definition inv_V {A : Type} {x y : A} (p : x = y) : p^^ = p :=
  match p with idpath => 1 end.

Definition concat_p_pp {A : Type} {x y z t : A} (p : x = y) (q : y = z) (r : z = t) :
  p @ (q @ r) = (p @ q) @ r :=
  match r with idpath =>
    match q with idpath =>
      match p with idpath => 1
      end end end.
Definition concat_pp_p {A : Type} {x y z t : A} (p : x = y) (q : y = z) (r : z = t) :
  (p @ q) @ r = p @ (q @ r) :=
  match r with idpath =>
    match q with idpath =>
      match p with idpath => 1
      end end end.
```

All of those should be understandable. Some may have alternate proofs. It is worth noting that many of these proofs are shorter than even the "second proofs" of the HoTT book.

The names seem unusual at first, but they follow the naming scheme of the HoTT Coq library:

- 1 means the identity path
- p means 'the path'
- V means 'the inverse path'
- A means '*ap*'
- M means the thing we are moving across equality
- x means 'the point' which is not a path, e.g. in *transport p x*
- 2 means relating to 2-dimensional paths
- 3 means relating to 3-dimensional paths, and so on

We'll see more functions named in this style as we proceed.

1.4.2 Functions are functors

Next, we define the function "transport" with its `Notation`.

```
Definition transport {A : Type} (P : A -> Type) {x y : A} (p : x = y) (u : P x) : P y :=
  match p with idpath => u end.
```

```
Notation "p # x" := (transport _ p x) (right associativity, at level 65, only parsing)
: path_scope.
```

Next comes the non-dependent and dependent versions of "ap". ("application of a function to a path" or "action across paths"). The HoTT Coq library calls the dependent version "apD" rather than "apd".

```
Definition ap {A B:Type} (f:A -> B) {x y:A} (p:x = y) : f x = f y
  := match p with idpath => idpath end.
```

```
Arguments ap {A B} f {x y} p : simpl nomatch.
```

```
Definition apD {A:Type} {B:A->Type} (f:forall a:A, B a) {x y:A} (p:x=y):
  p # (f x) = f y
  :=
  match p with idpath => idpath end.
```

```
Arguments apD {A B} f {x y} p : simpl nomatch.
```

In the HoTT book, the use of "ap" and "apd" is often implicit. The reader can determine when "f(p)" means "ap(f,p)" because "p" is a path and "f" called on a path just doesn't "fit" in the proof. In Coq, we have to be explicit about the use of "ap" and "apD".

1.4.3 Homotopy

So far, we've been covering types and functions in the same sequence as the HoTT book. At this point in the book there is the definition of "homotopy". But, if you've read further in the book, you know that homotopy and identity are equivalent. Thus, the HoTT Coq library has no need to define "homotopy" and neither do we.

We go straight to equivalences.

1.4.4 Equivalences

For equivalences, we need a definition of a "section" or the one-sided inverse to a function.

```
Definition Sect {A B : Type} (s : A -> B) (r : B -> A) :=
  forall x : A, r (s x) = x.
```

The actual definition of equivalence requires some new commands.

```
Class IsEquiv {A B : Type} (f : A -> B) := BuildIsEquiv {
  equiv_inv : B -> A ;
  eisretr : Sect equiv_inv f;
  eissect : Sect f equiv_inv;
  eisadj : forall x : A, eisretr (f x) = ap f (eissect x)
}.
Arguments eisretr {A B} f {_} _.
Arguments eissect {A B} f {_} _.
Arguments eisadj {A B} f {_} _.
Record Equiv A B := BuildEquiv {
  equiv_fun :> A -> B ;
  equiv_isequiv :> IsEquiv equiv_fun
}.
```

I'm going to address these commands from easiest to hardest, not first to last.

The easiest is the "Arguments" commands. Ignore them. They just define implicit arguments and we aren't covering the "Arguments" command in this tutorial.

Next, the `Record` command creates an inductive type with a single constructor. So, the type "Equiv" is very close to the dependent pair type "sigT". The constructor for the new type is "BuildEquiv". The `Record` command also creates projection functions for extracting the two elements stored in an "Equiv". These are called "equiv_fun" and "equiv_isequiv" (and are very similar to "projT1" and "projT2").

Lastly, we come to the `Class` command. `Class` operates similar to a `Record`, but it has special implicit argument rules. Thus, when Coq searches for an argument of type "IsEquiv f", it will look at all elements of that type declared with the "Instance" command. As a result, the second argument to "BuildEquiv" can often be left implicit.

The HoTT Coq library declares one default `Instance`, which is the second part of the "Equiv" record.

Existing `Instance` *equiv_isequiv.*

And, of course, there is a `Notation` for equivalence. There is also one for the inverse function inside it.

```
Notation "A <~> B" := (Equiv A B) (at level 85) : equiv_scope.
Notation "f ^-1" := (@equiv_inv _ _ f _) (at level 3) : equiv_scope.
Local Open Scope equiv_scope.
```

The definition used by "Equiv" is the "Half Adjoint Equivalence" of the HoTT book.

In the book, the parts of "ishae(f)" are (using Coq's notation):

- f : A -> B
- g : B -> A
- η : f o g ~~ idmap A
- ϵ : g o f ~~ idmap B
- τ : forall x:A, f (eta x) = epsilon (f x)

where "~~" represents homotopy, which is never defined in Coq. (We used two tildes since the single tilde ("~") in Coq is the "not" operator.)

In Coq, if we have a variable "e" of type "A <~> B", the parts are:

- equiv_fun e : A -> B
- equiv_inv (equiv_isequiv e) : B -> A
  - Also written: (equiv_fun e) ^-1 : B -> A
- eisretr (equiv_isequiv e): Sect equiv_inv f
- eissect (equiv_isequiv e): Sect f equiv_inv
- eisadj (equiv_isequiv e): forall x : A, eisretr (f x) = ap f (eissect x)

Notice that the final expression contains a call to "ap" that is implicit in the HoTT book's notation.

Now that we have a definition for equivalence, let's try to prove that it is an equivalence relation.

Properties of Equivalences

Our first example is proving reflexivity: that for all types "A", "ismap A" creates an equivalence between "A" and itself. Creating an equivalence usually takes three parts:

- creating an element of type Class "IsEquiv",
- registering it as an Instance (for implicit arguments), and
- creating the Record "Equiv".

The following command does the first two parts for reflexivity. "@BuildIsEquiv" creates the element of "IsEquiv" and the "Instance" command registers the element.

```
Instance isequiv_idmap (A : Type) : IsEquiv (idmap A) :=
  @BuildIsEquiv A A (idmap A) (idmap A) (fun _ => 1) (fun _ => 1) (fun _ => 1).
```

This next command creates the element of "Equiv" (written "A <~> A") by calling the constructor "BuildEquiv". As you can see, the final argument is inferred using the Instance registered by the previous command.

```
Definition equiv_idmap (A : Type) : A <~> A := @BuildEquiv A A (idmap A) _.
```

This looks like a lot of work to prove something that is obvious. And it is. However, the library is designed for proving more complex instances of equivalence. One aspect of that is once an "Instance" of "IsEquiv" is registered, it can be used as an inferred argument in many places. That won't happen for "isequiv_idmap", which is only use in a few places, but does happen.

After proving that equivalences are reflexive, we should prove that every equivalence has an inverse. However, that proof is rather long and complicated. (That has to do with the choice of half-adjoint equivalences; the proof for bi-invertible maps is just 8 lines.) Since this document is about reading what has been proven, and not the proofs themselves, we will cheat. We'll skip the proof and use the "admit" command so that you can read what has been proven.

```
Definition equiv_inverse : forall {A B : Type} (e : A <~> B), (B <~> A).
  admit.
Defined.
```

And here is what transitivity ("composition") looks like.

```
Definition equiv_compose' : forall {A B C : Type} (g : B <~> C) (f : A <~> B)
                                   , A <~> C.
  admit.
Defined.
```

It is called "equiv_compose"-prime, because there is a second function that instead takes functions from "A" to "B" and from "B" to "C" and uses the implicit arguments provided by `Instance` to build the equivalence.

1.4.5 Univalence

Since we've decided to cheat and add theorems without proof, it seems like the opportune time to add an axiom. Homotopy type theory's univalence axiom states that there exists an equivalence between two types being equivalence and the two types being equal.

```
Definition equiv_path (A B : Type) (p : A = B) : A <~> B :=
  match p with
    | idpath => equiv_idmap A
  end.

Class Univalence := {
  isequiv_equiv_path :> forall (A B : Type), IsEquiv (equiv_path A B)
}.

Instance univalence_axiom : Univalence.
  admit.
Defined.
```

The function "equiv_path" says that for every equality between types, there is an equivalence between them.

The `Class` "Univalence" says that for any two types, "equiv_path" determines an equivalence between the types. So, not only you can get a function that maps an equivalence to an equality, you know that that function is the inverse (with some qualifications) of "equiv_path".

Univalence Example

To end this document, we'll do a small proof. We'll declare a type that is the duplicate of "nat", prove they're equivalent, and then use the Univalence Axiom to conclude that they are equal.

First, we define our new type of *nat*s.

```
Inductive nat2 : Set :=
  | O2 : nat2
  | S2 : nat2 -> nat2.
```

Next, we define invertible maps between them. This is easy: we match zero to zero and successor to successor.

```
Fixpoint nat_to_nat2 (n : nat) : nat2 :=
  match n with
      | O => O2
      | S n' => S2 (nat_to_nat2 n')
  end.

Fixpoint nat2_to_nat (n2 : nat2) : nat :=
  match n2 with
      | O2 => O
      | S2 n2' => S (nat2_to_nat n2')
  end.
```

Next, we must prove that are maps are left- and right- inverses of each other. We could have done each in a single function, if we used "nat_rect" and "nat2_rect", but it is easier to read with the `match` expression.

```
Fixpoint sect_nat2_helper (x : nat2) : (nat_to_nat2 (nat2_to_nat x)) = x :=
  match x return (nat_to_nat2 (nat2_to_nat x)) = x with
    | O2 => idpath O2
    | S2 x' => ap S2 (sect_nat2_helper x')
  end.
Definition sect_nat2 : Sect nat2_to_nat nat_to_nat2 := sect_nat2_helper.
```

```
Fixpoint retr_nat2_helper (x : nat) : (nat2_to_nat (nat_to_nat2 x)) = x :=
  match x return (nat2_to_nat (nat_to_nat2 x)) = x with
    | O => idpath O
    | S x' => ap S (retr_nat2_helper x')
  end.
Definition retr_nat2 : Sect nat_to_nat2 nat2_to_nat := retr_nat2_helper.
```

Then, since we're using the half-adjoint equivalence, we need one of the coherences.

Although this document is not trying to teach you how to prove, it is worth pointing out that the following theorem is proved differently than the other ones. It uses Coq's "tactic language". The tactic language a large vocabulary of commands and multiple forms of automation to help prove theorems.

```
Theorem adj_nat2 (x : nat) : sect_nat2 (nat_to_nat2 x) = ap nat_to_nat2 (retr_nat2 x).
Proof.
  elim x.
    exact idpath.

    intros x' inductive_hyp.
    simpl.
    rewrite inductive_hyp.
    case (retr_nat2 x').
    exact idpath.
Qed.
```

Now that we have all 5 pieces needed for the equivalence, we make the "IsEquiv" element, register it as an `Instance`, and then make the "Equiv" element that witness that "nat" and "nat2" are equivalent.

```
Instance isequiv_nat_nat2 : IsEquiv nat_to_nat2 :=
  BuildIsEquiv nat nat2 nat_to_nat2 nat2_to_nat sect_nat2 retr_nat2 adj_nat2.
```

```
Definition equiv_nat_nat2 : nat <~> nat2 := BuildEquiv nat nat2 nat_to_nat2 _.
```

Next, we use the univalence axiom to build an equivalence between the equality of *nat* and *nat2* and the equivalence of *nat* and *nat2*.

```
Definition big_equiv : (nat = nat2 :> Type) <~> (nat <~> nat2) :=
  BuildEquiv (nat = nat2 :> Type) (nat <~> nat2) (equiv_path nat nat2) _.
```

With that equivalence, we can extract the inverse map, which takes the equivalence to the equality. Applying that function to the equivalence, gives us the equality. Thus, *nat* is equal to *nat2*!

```
Definition nat2_is_nat : (nat = nat2 :> Type) :=
  (big_equiv ^-1) equiv_nat_nat2.
```

With that type equality, we can take any theorem we've proved on *nat* and convert it into a theorem on *nat2*. In our final example, we'll convert the identity function on *nat*s into one on *nat2*.

```
Definition idmap_nat2 : nat2 -> nat2 :=
  match nat2_is_nat in (_ = y) return (y -> y) with
    | 1 => idmap_nat
  end.
```

1.5 Going Further

1.5.1 Homotopy Type Theory in Coq

The reference for HoTT in Coq is:

- http://homotopytypetheory.org/coq/

It contains links to the HoTT Coq library and proofs that use it. Thanks to this document, you should now be able to read what has been proven.

Also, the site contains links to the version of Coq that supports higher inductive types and is necessary to use the HoTT Coq library and to write new proofs using it.

1.5.2 General Coq references

The website for Coq is:

- http://coq.inria.fr/

Installation

The Coq website has links to compiled versions of standard Coq for Windows and OSX. If you're running Linux, many distributions have Coq available. Under Ubuntu and Debian, the command to install Coq and CoqIDE is "sudo apt-get install coq coqide".

CoqIDE is a graphical user interface for Coq. We strongly recommend using either CoqIDE or "Proof General", which lets you use Coq inside of the Emacs editor. (Available at http://proofgeneral.inf.ed.ac.uk/)

"ProofWeb" is a website that lets you interface to Coq by using a web browser. You will not need to install anything. It is available at http://prover.cs.ru.nl/ n

Documentation

A good introduction to Coq is "Software Foundations". It is, however, aimed at students studying programming languages. It does not get quickly to "how to prove".

- http://www.cis.upenn.edu/~bcpierce/sf/

The following is a good cheatsheet. Most importantly, it has a list of "Basic Tactics" that guides new users on what command to use when working with the powerful tactics language. Additionally, it has many of the book-to-Coq translations that are in this document.

- http://andrej.com/coq/cheatsheet.pdf

The Coq Reference Manual, with its explanations of every feature in standard Coq, is available at:

- http://coq.inria.fr/documentation