A Formal Introduction to Models of Computation in Coq

Reed Oei

April 12, 2019

Contents

1 Introduction

This series will work through some introductory material on models of computation. There are two goals: to teach the material itself, as well as teach how to use Coq/the basic theory behind it.

You should probably be somewhat familiar with some functional programming language, though which one is probably not important. You should also probably have some knowledge of basic math/logic, but other than that, it should be mostly self-contained.

This article will simply introduce the idea of using Coq to prove things, then doing some simple proofs involving strings to set things up for next time.

Note that the full text for all proofs are available at the GitHub repositroy: `https://github.com/ReedOei/FormalModelsOfComputation`.

All images of automata were generated by the website: `http://ivanzuzak.info/noam/webapps/fsm_simulator/`.

2 Introduction to Coq

[Coq](https://coq.inria.fr/) is a system for theorem proving, called a *proof assistant*. It includes several sub-programming languages for variable specifications of definitions and proofs.

2.1 Curry-Howard Isomorphism

The whole thing is based on the Curry-Howard isomorphism [1]; the basic idea is that *types are propositions, programs are proofs*. What does that mean?

Essentially, if it is possible to construct a value of some type, τ, then that type is "true", in the sense that we can produce a proof of it. And what is the proof? The value of type τ. This probably seems weird, because it's weird to think of the type `Int` as being "true", but it just means that I can prove that an `Int` exists: just pick your favorite one (mine is `0`).

In this case, a slightly more complicated example will probably make more sense. Let's consider implication: if P, then Q. What does this correspond to as a type? A function! Proving if P, then Q is the same as writing a function that takes a value of type P and produces a value of type Q. In Java we might write:

```
public <P, Q> Q f(P p) {
    // ...
}
```

but since this series is about Coq, I'll mostly write in Coq from now on:

```
Definition f : P -> Q := (* ... *).
(* alternatively *)
Definition f (p : P): Q := (* ... *).
```

As a more concrete example of implication, take the following introduction rule in intuitionistic logic [2].

$$\frac{A \quad B}{A \wedge B}$$

This rule says that if A is true, and B is true, then $A \wedge B$ is true. This is a fairly obvious definition for what "and" means, and it's a nice and simple example to demonstrate how the Curry-Howard isomorphism works. Recalling that "types are propositions", let's consider what the proposition here is: if A and B, then $A \wedge B$. What would this mean in a programming lanugage? If you have a value of type `a`, and a value of type `b`, then you can create a value which contains values of both type `a` and `b`. That is, a *pair*, of the two values.

In Haskell, we would write this as:

```
prop :: a -> b -> (a, b)
prop a b = (a, b)
```

The first line, the type, is the proposition. The second line, the definition of the function, is the proof: a program that shows how to go from the assumptions (a value of type `a` exists, and a value of type `b` exists), to the conclusion (so there is a value which contains values of both types).

Translating to Coq, this becomes:

```
Definition conjunc (a : A) (b : B) : A /\ B := conj a b.
```

3 Strings

3.1 Definining Strings

We will define strings inductively, because it's the most natural way to do it. In fact, we will actually going to define lists, not strings, but the two behave very similarly, at least for the purposes of this article.

```
Inductive list (A : Type) : Type :=
    | nil : list A
    | cons : A -> list A -> list A
```

This is actually already defined for us in Coq, so we'll just use the following lines to import it (and some nicer notation):

```
Require Import List.

Import ListNotation.
```

The new notation essentially gives us the following equivalences (note the below is not valid Coq code):

```
[] = nil
x :: xs = cons x xs
```

Similarly, we also want to use natural numbers (defined as follows), because we care about the length of our strings, and the length of a list is well-represented by a natural number.

```
Inductive nat : Type :=
    | O : nat (* note this is the letter O, not the number 0 *)
    | S : nat -> nat
```

Similar to above, we import a module to gain access to definitions, and, more importantly, proofs of obvious properties of natural numbers, like commutativity of addition. Without these proofs, it's a pain to work with any numbers because of the sheer quantity of seemingly "obvious" statements about natural numbers that we take for granted, such as associativity or commutativity of addition.

```
Require Import PeanoNat.
```

Finally, we define an arbitrary type, which essentially functions as the alphabet for our strings. That is, each "letter" (element) of our string (list) is a value of type `A`.

```
Variable A : Type.
```

3.2 Basic String Operations

Let's now define some basic operations: length, concatenation, and reversals.

Each of these definitions will be recursive, so we'll use the `Fixpoint` keyword to indicate this to Coq. Note that Coq requires *structural recursion*, which ensures that all our functions will terminate. Basically, structural recursion means that you can only do recursive calls on "smaller" values: if someone calls $f(n)$, you can recursively call $f(n-1)$, but not $f(n+1)$. Without this, nonterminating functions may be used to prove *anything*, which is not really what we want. We lose Turing completeness like this, but as you'll see, we can still prove plenty of things.

3.2.1 Length

The first operation to define is the length of a string. As you will see is quite common, we define the function for each *case* or *constructor* of the type we are recursing on; in this case, we are recursing on a list, so we need to cover both the empty list and the non-empty list. Doing this, we can define that a list has a length whose value is given by:

```
Fixpoint length (s : list A) : nat :=
    match s with
    | []      => 0
    | _ :: xs => 1 + length xs
    end.
```

That is, if we have the empty string, [] (or, mathematically ϵ), then it's length is just 0. Otherwise, if we have some symbol followed by a string xs, the total length is simply one greater than the length of xs. Note that when we don't care about the value of a particular variable, in this case, the first symbol in the string, we can simply write an underscore to indicate that we don't care. This is good practice because it's a hint to whoever is reading our definitions that they can mentally ignore that part altogether.

3.2.2 Concatenation

This operation will take two strings, "hello " and "world" and join them into "hello world". You are probably already familiar with this operation, but may not have seen the recursive definition before. Similar to above, we define it both for the case where `a` is the empty list, and when it is not empty. While recursive definitions like this may seem confusing, they actually make the proofs much more natural.

```
Fixpoint concat (a b : list A) : list A :=
    match a with
    | []      => b
    | x :: xs => x :: concat xs b
    end.
```

Note that, even though there are two strings, we still only recurse on one of them, specifically, the first one, `a`. We also introduce new notation to make our lives easier (this line allows us to write `x ++ y` instead of `concat x y`)):

```
Notation "x ++ y" := (concat x y).
```

3.2.3 Reversing

Finally, the reversal of a string. This definition is not computationally the most efficient (given our concatenation definition, it has to traverse the entire list to append the element to the end. We could fairly easily write a $O(n)$ version using an accumulator, but this definition is not only simpler to state, but it's also equivalent to the other definition. More importantly, given that we're just interested in it for proof purposes (for the moment, at least), we only care that it works and that it's easy to prove theorems with it. In this case, both criteria are satisfied, so we're good.

```
Fixpoint reverse (s : list A) : list A :=
    match s with
    | []      => []
    | x :: xs => reverse xs ++ [x]
    end.
```

As before, we handle the case of the empty list and non-empty list separately. Reversing an empty list has no effect. Reversing a non-empty list simply takes the first element, reverses the rest of the elements, and places this first element at the end of the list. If you are unsure of how this definition (or any of the preceding definitions) works, you should stop and take the time to figure them out.

3.3 Proving Basic String Properties

3.3.1 Empty String is Concatenation Identity

The first of these is that the empty string, ϵ functions as a right identity for concatentation (it is already a left identity by definition). We write the following lines, which put us into Coq's interactive proof evironment. Here we use a set of *proof tactics* which take our current propositions and transform them until we can apply axioms to finish the proof.

```
Lemma concat_empty_string_id : forall (s : list A), s ++ [] = s.
Proof.
```

After this line, the goals we have are:

```
forall (s : list A), s ++ [] = s.
```

This is exactly what we wrote, and to start the proof, we must *introduce* variables, so that we can talk about the variables in our hypothesis. In this case, we have only one variable, `s`. Continuing the idea that proofs are functions, we can think of this variable as a parameter to the function, `concat_empty_string_id`‘ To add this to our goals, we can write:

```
intro s.
```

Now we have:

```
1 subgoal
s : list A
______________________________________(1/1)
s ++ [] = s
```

Above the bar is our hypotheses, and below the line is what we are trying to prove. This means we are trying to prove, for some arbitrary list `s`, that `s ++ [] = s`. We will do so by structural induction on the list (this is one of the things defined for us in the `List` module). Again, going back to programming, this is like case anaylsis: if the list is empty, then we our function. Otherwise, if the list is not empty, we write our function by recursively calling our function on the smaller list. To use induction, we simply write:

```
induction s.
```

Now we get:

```
2 subgoals
______________________________________(1/2)
[] ++ [] = []
______________________________________(2/2)
(a :: s) ++ [] = a :: s
```

This means that we have two *subgoals* to prove: the base case, when the list is empty, and the inductive case, when it is not. In this inductive case, we know that the list of is of the form `a :: s`, that is, there is some element, `a`, followed by possibly many more characters, that is, another string, called `s`.

By default, we prove the subgoals in order, though it is possible to switch the order if you really need to. In this case (and most), the default order works well. Thinking about what the first subgoal is saying, `[] ++ [] = []`, we know that it is quite obviously that this is true—but we must still provide a proof. Luckily, simply by definition (in the first case), we can see that the left hand side reduces to `[]` (this is the base case in our definition of the `++` operation). In Coq, to apply function definitions, we can use the `simpl` tactic, like so:

```
simpl.
```

Now we are left with:

```
2 subgoals
______________________________________(1/2)
[] = []
______________________________________(2/2)
(a :: s) ++ [] = a :: s
```

Whenever you see the same thing on both sides of the equality, you can use the `reflexivity` tactic to prove your goal, since the proposition is true by reflexivity. So we end the proof of this case with:

```
reflexivity.
```

This resolves the first subgoal, and now we are left with:

```
1 subgoal
a : A
s : list A
IHs : s ++ [] = s
______________________________________(1/1)
(a :: s) ++ [] = a :: s
```

This looks considerably more complicated, so let's consider each part on it's own. By doing induction, we've handle the case of the empty string, so now we just need to handle the case that the string is not empty, that is, it starts with some symbol `a` and ends in a string `s`. The part below the line is simply the thing we are trying to prove, `x ++ [] = x`, but with `a :: s` instead of `x`. Finally, `IHs` is the inductive hypothesis, which tells us that the lemma is true for smaller list, `s`.

Usually, if we have an equality, we will want to make a substitution in our goal, either substituting the left hand side for the right, or vice versa. However, in this case, substitution doesn't seem to help us, as the only way we can substitute, the right hand side for the left, only serves to complicate the equation. Instead, we want to simplify the equation first. It's often a good idea to simplify your equation whenever you can, and in this case, we can apply the second case in our `concat` function (via the `simpl` tactic) to simplify this as follows.

```
1 subgoal
a : A
s : list A
IHs : s ++ [] = s
______________________________________(1/1)
a :: (s ++ []) = a :: s
```

I have added the parentheses myself for clarity. Now we can use the inductive hypothesis, `IHs` to replace `s ++ []` with `s`. To do so, we use the `rewrite` tactic, like so:

```
rewrite IHs.
```

Note that to substitute the right hand side for the left, that is, to substitute `s ++ []` for `s`, we would write:

```
rewrite <- IHs.
```

After rewriting, we have:

```
1 subgoal
a : A
s : list A
IHs : s ++ [] = s
______________________________________(1/1)
a :: s = a :: s
```

Now we have an equality with both sides the same, and we can finish the proof with an application of `reflexivity`. When you see:

```
No more subgoals.
```

When you see this, you can finish the proof with:

```
Qed.
```

The entire proof is shown below, with the tactics for each induction subgoal on the same line:

```
Lemma concat_empty_string_id : forall (s : list A), s ++ [] = s.
Proof.
intuition.
induction s.
simpl. reflexivity. (* empty list case *)
simpl. rewrite IHs. reflexivity. (* non-empty list (inductive) case *)
Qed.
```

The rest of the proofs will be done in far less detail.

3.3.2 Length with Concatenation

The next lemma is that the length of `x` concatenated with `y`, is the length of `x` added to the length of `y`. In Coq we write:

```
Lemma length_concat : forall (a b : list A), length (a ++ b) = length a + length b.
```

The proof is nearly identical to the first proof we did, except that we only do induction on `a`. This makes sense, given that concatenation is defined recursively only on the first argument. We begin as we did before:

```
Proof.
intros a b.
induction a.
```

The base case is again obvious by simplification:

```
simpl. reflexivity.
```

The other case is proved similarly:

```
simpl. rewrite IHa. reflexivity.
```

Note that we use `IHa` rather than `IHs`, because we did induction on `a`, rather than `s`. Had we named our variables differently, the inductive hypothesis will have a different name.

3.3.3 Concatenation is Associative

This lemma is an important fact that is very useful for proving things, as well as essentially finishing the proof that strings form a monoid under concatenation. In Coq:

```
Lemma concat_assoc : forall (a b c : list A), a ++ (b ++ c) = (a ++ b) ++ c.
```

The proof is identical to the previous two (but you should still do it!).

3.4 Various String Lemmas

In this section we continue to prove several more lemmas involving the definitions and lemmas above.

3.4.1 Reversing a string doesn't change it's length

```
Lemma length_reverse : forall (s : list A), length s = length (reverse s).
```

This proof will require slightly more effort, though we begin with induction just like the other proofs. In particular, we'll need the use of a lemma from above and the fact that addition is commutative for natural number. Beginning by induction, the base case is trivial because the length of an empty string is 0 by definition, and the reversal of an empty string is also an empty string.

```
intros s.
induction s.
simpl. reflexivity.
```

Our inductive case is now to show (after a simplification):

```
S (length s) = length (reverse s ++ [a])
```

S denotes the successor function, so our goal is $1 + |s| = |s^R \cdot a|$, where s^R is the reverse of s. Luckily, we have a lemma from above that allows us to simplify the right hand side (that is, $|x \cdot y| = |x| + |y|$. We can apply it like so, and then applying the inductive hypothesis almost finishes the proof.

```
rewrite length_concat.
simpl.
rewrite IHs.
```

The final step we need is to rewrite $|s^R| + 1$ as $1 + |s^R|$, which can be accomplished by a built-in theorem, imported from `PeanoNat`. This property is simply commutativity of addition, and is appropriately named `Nat.add_comm`. After application, simplification finishes the proof for us:

```
rewrite Nat.add_comm.
simpl.
reflexivity.
Qed.
```

3.4.2 Reversing Concatentation

```
Lemma reverse_concat : forall (a b : list A), reverse (a ++ b) = reverse b ++ reverse a.
```

As all the other proofs in this article, we begin via induction:

```
intros a b.
induction a.
simpl.
```

This time, however, the base case is not as trivial. We must show $b^R = b^R \cdot \epsilon$, which is obviously true, but is not evident from the definition. Luckily, we proved this earlier, so we can reuse our proof, which finishes up this case:

```
rewrite concat_empty_string_id.
reflexivity.
```

For the inductive step, we immediately simplify and use the inductive hypothesis. This makes our goal into:

```
(reverse b ++ reverse a0) ++ [a] = reverse b ++ reverse a0 ++ [a]
```

The only difference here is the grouping—luckily we proved that concatenation is associative earlier, so here we simply use it:

```
rewrite <- concat_assoc.
reflexivity.
Qed.
```

Note that we rewrite the "opposite" way: replacing the right hand side with the left, so we use an arrow ¡-. This finishes our proof.

3.4.3 Reversing a reversed string gives the original string

```
Lemma reverse_reverse_id : forall (s : list A), reverse (reverse s) = s.
```

Again, we proceed by induction. Here the base case is trivial, as it all simplifies by simple application of the definitions:

```
intros s.
induction s.
simpl. reflexivity.
```

The inductive case is also relatively straightforward with an application of the previous theorem, followed by the use of the inductive hypothesis.

```
simpl.
rewrite reverse_concat.
simpl.
rewrite IHs.
reflexivity.
Qed.
```

3.4.4 Proof Simpliciation with Intuition

As a parting note, the tactic `intuition` can often simplify multiple steps at a time, as well as serving to introduce variables and hypotheses. So we can rewrite the first proof we did as, though in this case, it provides only a minor benefit:

```
Lemma concat_empty_string_id : forall (s : list A), s ++ [] = s.
Proof.
intuition.
induction s.
intuition.
simpl. rewrite IHs. reflexivity.
Qed.
```

Also, note that while we wrote our proofs using proof tactics with the interactive proving environment, we can also write proofs as normal functions (it just takes more work).

4 Deterministic Finite Automata

In the previous section, we discussed working with Coq and strings/lists. This time we'll move onto defining and proving some interesting theorems about one of the simplest models of computation: deterministic finite automata, often abbreviated DFAs.

4.1 What is computation?

But before we do that, we discuss what we mean by "computation."

One of the simplest definitions, and the one that we'll be using, at least for now, is the following:

Computation is the process of determining whether a string is in a language or not.

This raises many more questions, such as "what is a language?", and the more fundamental "what does 'determining' mean?", both of which are indeed very important and will be addressed shortly. First, however, we will try to give some intuition into why this is a useful definition of computation.

Let's consider some simple computational problem, namely: Given a string w, does it have an even or odd number of characters? Because this is a boolean decision, we can simply the problem a little more: Given a string w, does it have an even number of characters? Clearly this requires some computation to check. In your favorite programming language (which is Haskell, of course), we could solve this problem as follows:

```
evenLen str = length str \mintinline{coq}{mod} 2 == 0
```

This definition is concise and familiar to any programmer, but it requires the definition of several more terms, and is therefore harder to prove theorems about. Therefore, it's sometimes preferable define `evenLen` as follows:

```
evenLen ""       = True
evenLen [_]      = False
evenLen (_:_:xs) = evenLen xs
```

Professional programmers might be offended by the runtime, but we can prove the two definitons are equivalent and that's all we care about at the moment.

But there are other computational problems that don't seem to fit this mold. For example, one of the most basic computations: compute $\text{inc}(x) = x + 1$ for any x. But take a step back: this is still a string recognition problem! Just think about it as a string, instead of an expression. For example, "f(2) = 3" is a valid string, but "f(4) = -1" is not. How would we check this? Well, one way would be as follows:

```
incString x str = str == ("f(" ++ show x ++ ") = " ++ show (x + 1))
```

You can imagine that any algorithm that recognizes this language must somehow "do addition" in the background.

Now we have some intuitive sense of what computation is, but no sense of what the "rules" are, so to speak. Clearly there are some things we can compute, and there are things we can't compute. For example, as you may know, most real numbers are **[cite]**. But without defining precisely what a computation is allowed to do, it's quite difficult to say whether a given language is computable or not.

4.2 Definitions

Now let's actually define what we mean by "language" and define the first model of computation, DFAs.

Defining a language is quite simple: a *language* is a set of *strings*. Note a language may be empty, and also may be infinite: in fact, infinite languages are generally much more interesting that finite languages. Any finite language is, from a purely mathematical standpoint, trivial to compute: given a string, we can just iterate through each string in the language and check, because the language is finite.

Now let's talk about DFAs. Here is an example of a DFA:

DFAs, also sometimes called state machines, are a very simple model of computation, which are generally draw as directed graphs. The nodes are called *states* and the edges are called *transitions*. One of the states is designated as a *start state*, and some possibly empty subset of the states are *final* or *accepting* states, indicated by two concentric circles.

To process an input string, you start at the start state, the move through the string, one character at a time, following the transition labeled with the current character. If the state you end up in when you are done processing the string is a final state, then the string is said to be *accepted*.

For example, the DFA above will accepted the string "abab": we start in state 0, then follow the arrow for "a" to state 1, the arrow for "b" to 0, the arrow for "a" to 1, and finally the arrow for "b" to "0", which has two circles, indicating it is a final state. Note that this DFA will reject the string "aa": we start in state 0, then follow the arrow for "a" to state 1, and then the arrow for "a" to state 2, which is not a final state. This DFA accepts strings that look like "abababababab...", and rejects everything else.

Formally, we define a DFA M as a five-tuple: Q,
$Sigma$,
$delta, s, F$, where: - Q is the set of states. Note that this set must be **finite** (hence deterministic **finite** automaton). - Σ is the *alphabet*, or the set of characters that are allowed to appear in the input strings. This is often omitted when it is obviously from the context. - $\delta : Q \times \Sigma \to Q$ is the transition function, which tells you, given a state and a character, what the next state should be. For example, in the DFA above, we have $\delta(0, a) = 1$ and $\delta(1, b) = 0$. - $s \in Q$ is the start state. - $F \subseteq Q$ is the set of final states.

Finally, for convenience we define the function $\delta^* : Q \times \Sigma^* \to Q$ as follows. Note that Σ^* is the set of all (finite) strings that can be formed from the alphabet, or set of symbols, Σ.

$\delta^*(q, \epsilon) = q$

$\delta^*(q, x \cdot w) = \delta^*(\delta(q, x), w)$

where $x \cdot w$ denote the string formed by the symbol x followed by the string w. Essentially, this function performs the same function as δ, but for a whole string.

Finally, we say a string w is *accepted* by a DFA M if $\delta^*(s, w) \in F$. The language defined by M is the set of strings accepted by M, and is defined as

follows:

$L(M) := \{w \in \Sigma^* : w \text{ is accepted by } M\}$

We say a language L is recognized by a DFA M if $L = L(M)$, and naturally that a language L can be recognized by a DFA if $L = L(M)$ for some DFA M.

4.3 Coq Definitions

So how do we define these concepts in Coq?

First, let's start by considering how to define a set. We could define a set as a list, then define equality that doesn't care about order, and proceed in a standard set theoretic way.

However, there's a far simpler definition: types and sets are very similar in many ways. While there are differences, we don't need to concern ourselves with those now. Both have values which are "members" of them, for sets, with the relation $x \in A$ and for types with the relation $x : A$. This is the property that we care about, so we'll call a type a "set" for now. In the future, we will need to restrict our definition to only allow finite sets.

4.3.1 Defining a DFA

So to define a DFA we need a set Q, a set Σ, a function δ, which we'll call `t`, for **t**ransition, a start state s, and a set of final states F. We said that sets will be represented by types, so we'll let Q and Σ each be the types `A` and `B`, respectively. Coq has support for functions, so we'll have a function `t : A -> B -> A` (the curried version of `t : A * B -> A` where `A * B` denotes the type of pairs of values of type `A` and `B`. The start state is simply some value of type `A`, so `s : A`. Finally, we have the set of final states. While this is a set, it's not just any set—it's a subset of `A`. We'll represent this by a function `F : A -> bool`. If `F q` is a true, then `q` is a final state, otherwise `q` is not a final state. Putting this all together we get:

```
Structure dfa (A B : Type) := {
  t : A -> B -> A;
  s : A;

  F : A -> bool;
}.
```

Here we have defined a *structure*, which is essentially just a record, or a product type with named projections. Note that it is parameterized over the types representing the set of states Q and the alphabet Σ.

To access some part of a dfa `M : dfa A B`, for example, the start state, we can write `t A B M`. It is quite tedious to always write the types `A` and `B`, and they can be easily inferred from the fact that `M` is one of the parameters. Therefore, we make use of `Arguments` command to make the types `A` and `B` implicit parameters:

```
Arguments t {A} {B}.
Arguments s {A} {B}.
Arguments F {A} {B}.
```

4.3.2 Defining languages

We now with to define δ^*, which we'll call `tStar`, and the language $L(M)$. Note that `tStar` has a nice recursive definition, so we'll define it as a `Fixpoint`, similar to how we defined some functions last time

```
Fixpoint tStar {A B : Type} (M : dfa A B) (q : A) (str : list B) : A :=
  match str with
  | []      => q
  | x :: xs => tStar M (t M q x) xs
  end.
```

Note that we make `A` and `B` implicit parameters of this function as well, by writing them inside the curly brackets, `{}`.

We define whether a string `str` is accepted by a DFA `M` is accepted by using the function `tStar` and the function `F` defined for every DFA as follows:

```
Definition accepted {A B : Type} (M : dfa A B) (str : list B) : bool :=
  F M (tStar M (s M) str).
```

Then the language defined by `M` is all `str : list B` such that `accepted M str = true`.

4.4 What languages can be recognized by a DFA?

This is a complicated question, but we can say some things with little effort.

For example, for an alphabet Σ, the language Σ^* is easily recognizable by a DFA. For example, if Σ^* is $\{a, b\}$, the following DFA does what we want:

Similarly, the empty language, $\{\}$ is also easily recognizable by a DFA.

What about other languages? Well, we can also recognize the set of strings with even length, for any alphabet Σ, using an automaton similar to the following:

We won't verify any of these in Coq, but you can convince yourself that they do in fact work.

Let's consider some more general questions of recognizability:

For example, if L_1 and L_2 are recognizable by some DFAs M_1 and M_2, respectively, is $L_1 \cap L_2$? What about $L_1 \cup L_2$? What about $\overline{L_1}$?

It's probably good to spend some time on your own thinking about whether or not these languages are always, sometimes, or never recognizable by a DFA. Keep in mind it must be a **single DFA** processing the languages in a **single pass**.

Note that both DFAs operate over the same alphabet.

4.5 DFA Closure Properties

As it turns out, the answer to all of these is "Yes, always!" These properties are sometimes referred to as "closure properties."

4.5.1 DFA Complement

Let's consider the problem of defining a DFA $\overline{M}$ given a DFA M, so that $L(\overline{M}) = \Sigma^* \setminus L(M)$, that is, $\overline{M}$ recognizes the complement of M.

This is actually quite simple, we just need to "flip" the final states: if $q \in F$ is a final state, then $q \notin \overline{F}$, and if $q \notin F$ then $q \in \overline{F}$. Everything else is the same!

Let's define this new set of final states, using the `negb` function, where `negb true = false` and `negb false = true`, as you would expect.

```
Definition not_dfa_f {A B : Type} (M : dfa A B) (q : A) : bool := negb (F M q).
```

Then we can define the DFA $\overline{M}$, which we will call `not_dfa M`, to be slightly more descriptive, as follows:

```
Definition not_dfa {A B : Type} (M : dfa A B) : dfa A B :=
  Build_dfa A B (t M) (s M) (not_dfa_f M).
```

This definition says that everything that we use the same types `A` and `B`, the same transition function `t M`, the same start state `s M`, and only change the set of final states from `F` to `not_dfa_f M`.

Proof of correctness Hopefully it makes intuitive sense that this is the correct way to define $\overline{M}$, but now let's prove it in Coq.

We can state the theorem that this definition is correct as follows:

```
Theorem not_dfa_correct :
  forall {A B : Type} (M : dfa A B) (str : list B),
   accepted M str = true <-> accepted (not_dfa M) str = false.
```

Mirroring Lemma Before proving this, however, we'll prove that $\overline{M}$ "mirrors" M: if after processing the string w with M we are in state q, then after processing the same string w with $\overline{M}$ we are also in state q. This lemma makes proving the above theorem quite simple. We can state the lemma as follows:

```
Lemma not_dfa_mirror {A B : Type} (M : dfa A B) :
  forall (str : list B), tStar M (s M) str = tStar (not_dfa M) (s M) str.
```

However, we will state and prove yet one more lemma before we prove the statement above. We'll need a very useful fact about δ^* that should be obvious from the definition:

$\delta^*(q, w \cdot x) = \delta(\delta^*(q, w), x)$

We can write this lemma in Coq as follows:

```
Lemma tStar_step :
  forall {A B : Type} (M : dfa A B) (str : list B) (q : A) (x : B),
    tStar M q (str ++ [x]) = t M (tStar M q str) x.
Proof.
```

As always, we first introduce some variables:

```
intros A B M str.
```

As with most proofs about lists, we proceed by induction on the list. The base case is easily solved by `intuition`.

```
induction str.
intuition.
```

Note that we don't introduce variables for `q` and `x` because that would over specialize our induction hypothesis to the point where we couldn't prove our goal anymore. After proving the base case, which is still easily true, we would get the following. Note that the below is not the entire state, only the part that's relevant.

```
IHstr : tStar M q (str ++ [x]) = t M (tStar M q str) x
______________________________________(1/1)
tStar M (t M q a) (str ++ [x]) = t M (tStar M (t M q a) str) x
```

Let's think about what this is saying: if we start from some state `q`, then process `str ++ [x]`, we ge the same thing as if we started from `w`, processed `str`, and now process `x`. That's true, of course, and so is the goal, but the goal doesn't directly follow from the inductive hypothesis.

Instead, by not introducing `q`, we obtain the following more general induction hypothesis:

```
IHstr : forall (q : A) (x : B),
        tStar M q (str ++ [x]) = t M (tStar M q str) x
```

Then we may simply utilize the inductive hypothesis with `q` being `t M q a` (note these `q` are different), as follows:

```
exact (IHstr (t M q a) x).
```

Now let's prove the mirroring property from above:

```
Lemma not_dfa_mirror {A B : Type} (M : dfa A B) :
  forall (str : list B), tStar M (s M) str = tStar (not_dfa M) (s M) str.
Proof.
```

Again, we will proceed by induction, but in a slightly different fashion.

A standard induction proof of some property `P` for lists is generally as follows:

- Prove that `P []` holds. - Prove that if `P xs` holds, then `P (a :: xs)` holds, for any `a`.

However, consider what this means for our property. The base case works fine, but the inductive case causes issues:

We want to show that if $\delta^*(s, w) = \overline{\delta}^*(s, w)$, then for any $x \in \Sigma$, we have $\delta^*(s, x{\cdot}w) = \overline{\delta}^*(s, x{\cdot}w)$. Of course, this is true, but we have no information about what state we end up in after processing x! Whereas, we do know something about the state $\delta^*(s, w)$. We want to show the following instead: if $\delta^*(s, w) = \overline{\delta}^*(s, w)$, then for any $x \in \Sigma$, we have $\delta^*(s, w{\cdot}x) = \overline{\delta}^*(s, w{\cdot}x)$. This is equivalent, but much easier to prove.

To use this form of induction, we use the lemma `rev_ind` defined in the Coq list library:

```
apply rev_ind.
```

Note that Coq can infer we want to do induction on `str` because of our goal:

```
forall str : list B, tStar M (s M) str = tStar (not_dfa M) (s M) str
```

The base case can be proven by `intuition`. In the inductive case after introducing variables, we have the following. Note that some of the unimportant details have been omitted, so your screen won't look identical.

```
H : tStar M (s M) l = tStar (not_dfa M) (s M) l
______________________________________(1/1)
tStar M (s M) (l ++ [x]) = tStar (not_dfa M) (s M) (l ++ [x])
```

This is a perfect place to use our lemma `tStar_step` from before! Let's use it twice: once for the left hand side, and once for the right hand side:

```
rewrite tStar_step.
rewrite tStar_step.
```

Coq can infer what we want to do, so we don't need to write anything else. In fact, when we do multiple rewrites in a row, we can instead write:

```
rewrite tStar_step, tStar_step.
```

In fact, we could even use the tactic `repeat` to make this simpler, if we didn't know how many rewrites we wanted (though I'll leave it as the above):

```
repeat (rewrite tStar_step).
```

Now we can simplify to get:

```
H : tStar M (s M) l = tStar (not_dfa M) (s M) l
______________________________________(1/1)
t M (tStar M (s M) l) x = t M (tStar (not_dfa M) (s M) l) x
```

Note that now we can use the inductive hypothesis to rewrite the two sides to be the same (it doesn't matter which way we rewrite), letting us finish the proof as follows:

```
simpl.
rewrite H.
reflexivity.
Qed.
```

Main Theorem Now we can finally prove the main theorem, about the correctness of our definition of $\overline{M}$:

```
Theorem not_dfa_correct :
  forall {A B : Type} (M : dfa A B) (str : list B),
   accepted M str = true <-> accepted (not_dfa M) str = false.
Proof.
```

We start by introducing our variables and unfolding the definition of accepted, because we essentially want to prove that the final function was defined correctly. To do so, we will use the fact that the two DFAs mirror each other.

```
intros.
unfold accepted.
simpl.
rewrite not_dfa_mirror.
unfold not_dfa_f.
```

This gives us the following goal:

```
F M (tStar (not_dfa M) (s M) str) = true <->
negb (F M (tStar (not_dfa M) (s M) str)) = false
```

We will prove each direction separately; we will do the -> direction first. To do this, we use the `split` tactic:

```
split.
```

which gives us the following:

```
H : F M (tStar (not_dfa M) (s M) str) = true
______________________________________(1/2)
negb (F M (tStar (not_dfa M) (s M) str)) = false
```

By rewriting with `H`, we obtain `negb true = false`: this is true by `intuition`.

For the other direction, <-, we have:

```
H : negb (F M (tStar (not_dfa M) (s M) str)) = false
______________________________________(1/1)
F M (tStar (not_dfa M) (s M) str) = true
```

Because our values are booleans, it is easy to do case analysis. To do this, we use `induction` on the boolean value. 'induction' works for any inductively defined type, including booleans which are defined as follows:

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

Then each case follows by `intuition`:

```
induction (F M (tStar (not_dfa M) (s M) str)).
intuition.
intuition.
Qed.
```

This completes the proof, showing that $\overline{L(M)}$ is a language recognized by a DFA. For example, this means that, if we can define a DFA which recognizes strings of even length, we may use the above definition to define an automaton which recognizes strings of odd length.

4.5.2 DFA Conjunction

So how do we show that we can recognize the language $L_1 \cap L_2$, given that both languages are recognized by DFAs? Essentially, we want to process these strings in "parallel", working in both DFAs at the same time? We can do this by keep track of the state in M_1 and the state in M_2 separately, as follows, using a new DFA, M':

- The set of states is the cross product of the states of M_1 and M_2, $Q_1 \times Q_2$. - The alphabet is the same, because we want to be able to process all the same strings. - The transition function is essentially just a combination of the two transition functions. Recalling that our states are now pairs of the states in the old DFAs, we can define δ^* as follows:

$\delta'((q_1, q_2), x) = (\delta_1(q_1, x), \delta_2(q_2, x))$

- The start state is naturally just the pair of the start states of the DFAs, (s_1, s_2). - The set of final states is the cross product of the sets of final states from the DFAs: $F_1 \times F_2$.

This is sometimes referred to as

So why does this work?

Intuitively, it's because we run the two DFAs in parallel—the first value in the pair keeps track of the state in the first DFA, and the second value keeps track of the state in the second DFA. Of course, we can do better than that, and show that it's true using Coq.

Definitions It's probably good practice to try to define `and_dfa M N` for twice DFAs `M` and `N` in Coq on your own before reading the definition below.

Hopefully the below Coq definition is fairly intuitive by now:

```
Definition and_dfa_trans
  {A B C : Type} (M : dfa A B) (N : dfa C B) (q : A * C) (s : B) : A * C :=
```

```
  match q with
  | (qm, qn) => (t M qm s, t N qn s)
  end.

Definition and_dfa_f {A B C : Type} (M : dfa A B) (N : dfa C B) (q : A * C) : bool :=
  match q with
  | (a,c) => F M a && F N c
  end.

Definition and_dfa {A B C : Type} (M : dfa A B) (N : dfa C B) : dfa (A * C) B :=
  Build_dfa (A * C) B
            (and_dfa_trans M N) (s M, s N) (and_dfa_f M N).
```

Mirroring Proof We also prove a similar mirroring property for our definition of the `and_dfa`:

```
Lemma and_dfa_mirror_m
  {A B C : Type} (M : dfa A B) (N : dfa C B) :
    forall (str : list B), tStar M (s M) str = fst (tStar (and_dfa M N) (s (and_dfa M N)) st
```

Again we proceed by induction, specifically, using `rev_ind` as before, solving the base case of an empty string by `intuition`:

```
apply rev_ind.

intuition.

intros.
```

Next we use the lemma `tStar_step` twice again, for the left hand side and right hand side:

```
rewrite tStar_step, tStar_step.
```

Now we have:

```
H : tStar M (s M) l = fst (tStar (and_dfa M N) (s (and_dfa M N)) l)
______________________________________(1/1)
t M (tStar M (s M) l) x =
fst (t (and_dfa M N) (tStar (and_dfa M N) (s (and_dfa M N)) l) x)
```

By our definition of `t` for `and_dfa`, `tStar (and_dfa M N) (s (and_dfa M N)) l` will be a pair, so we use `destruct` on it to get access to this pair, which will be called `(a,c)`. That is, $\delta'^*((s_m, s_n), \ell) = (a, c)$. We also want to `unfold` the definition of `and_dfa` so we can use the definition of `t` which is "inside" of it:

```
destruct (tStar (and_dfa M N) (s (and_dfa M N)) l).
unfold and_dfa.
simpl.
```

Now we have:

```
H : tStar M (s M) l = fst (a, c)
______________________________________(1/1)
t M (tStar M (s M) l) x = t M a x
```

By rewriting the inductive hypothesis, this is obviously true by the definition of `fst`, and `intuition` can solve this for us, finishing the proof.

```
rewrite H.
intuition.
Qed.
```

Note the proof for the second component is identical, except with `snd` and `N` instead of `fst` and `M`.

Main Theorem First we must define

We state the theorem similarly to the previous theorem about complements:

```
Theorem and_dfa_mirror_m
  {A B C : Type} (M : dfa A B) (N : dfa C B) :
    forall (str : list B), tStar M (s M) str = fst (tStar (and_dfa M N) (s (and_dfa M N)) st
Proof.
```

We introduce our variables and `unfold accepted` again. Then using our mirroring properties from above, our goal becomes:

```
F (and_dfa M N) (tStar (and_dfa M N) (s (and_dfa M N)) str) = true <->
F M (fst (tStar (and_dfa M N) (s (and_dfa M N)) str)) &&
F N (snd (tStar (and_dfa M N) (s (and_dfa M N)) str)) = true
```

This is a bit of a mess, but what's important is that all of these "`F`s" are applied to the same thing: `tStar (and_dfa M N) (s (and_dfa M N)) str`. We can destruct this, as shown below:

```
destruct (tStar (and_dfa M N) (s (and_dfa M N)) str).
simpl.
```

Giving us the following goal:

```
F (and_dfa M N) (a, c) = true <-> F M (fst (a, c)) && F N (snd (a, c)) = true
```

We could do the case analysis ourselves—but luckily so can `intuition`:

```
intuition.
Qed.
```

This concludes our proof that our new DFA really does recognize the language $L_1 \cap L_2$.

4.6 Remarks

Note that we have shown above that $L_1 \cap L_2$ and $\overline{L}$ are recognizable by a DFA. However, this implies that both the languages $L_1 \cup L_2$ and $L_1 \implies L_2$ are also recognizable. We define $L_1 \implies L_2$ as below:

$L_1 \implies L_2 := \{w \in \Sigma^* : w \in L_1 \implies w \in L_2\}$

The following are also true:

$L_1 \cup L_2 = (\overline{\overline{L_1} \cap \overline{L_2}})$

$L_1 \implies L_2 = \overline{L_1} \cup L_2$

The reader may check that the above equalities hold. Since we can define $\cup$ and $\implies$ in terms of $\cap$ and complements, we can also recognize unions and implications.

4.7 Conclusion

We have shown some fairly general properties languages which are recognizable by DFAs. In the next part, we will show an alternate model of computation, called a non-deterministic finite automaton, that turns out to be quite useful, and prove some surprising facts about it.

5 Nondeterministic Finite Automata

In the previous section, we talked about our first model of computation, the deterministic finite automaton, or DFA. It was very simple, but we can see that it is capable of some computation, and furthermore, they have nice closure properties that make working with them easier.

This time, we'll talk about a new model of computation, learn about some new Coq features, and prove theorems about Coq programs, rather than just about our own definitions.

You may be wondering why we care to specify that DFAs are deterministic; after all, that tends to be the default in mathematics. What would a *nondeterministic* finite automaton look like?

Well, wonder no longer! Or maybe do, as it probably instructive to attempt to think about what it would mean, before reading on.

5.1 More Powerful Models of Computation

5.1.1 Are there languages not recognized by a DFA?

Before we ask the question, "What's more 'powerful' than a DFA?", whatever "powerful" means, we should ask: "Is there anything more 'powerful' than a DFA?" And we'll start by defining what we mean by "powerful". If you recall, we defined computation as the process of deciding whether a string is in a language or not; that is, defining a language, or set of string. We can say that a model of computation can recognize some set of languages (that is, all languages recognized by any DFA). Then a more powerful model of computation would

be one which can decide all the languages that a DFA can, **and** at least one other languages.

Let's consider whether there are **any** languages that cannot be defined by a DFA. We'll formalize this later, but the short answer is "yes". Intuitively, this probably makes sense: DFAs only have a finite amount of "memory", in the form of states.

For example, how would you build a DFA that recognizes whether the length of a string was some prime number? Of course, you can do this for any *particular* prime number (just make a DFA with as many states as you wish, and advance by one state each time); the problem comes with trying to do it for *every* prime number. This intuitively requires us to be able to do arbitrary precision arithmetic. Your computer needs arbitrarily large amounts of memory to do this, so how would a DFA be able to do it with a finite amount of memory? Again, we'll formalize this later, but for now hopefully you're convinced that **some** languages are not recognizable by a DFA.

5.1.2 Can we do better than DFAs?

So there are some languages not recognizable by a DFA.

Let's try to do better, but keep it simple so we can still prove things about it. For example, while your laptop may be capable of computing basically anything, we don't want to be simulating the entirety of a modern CPU in Coq. Instead, let's start out much more modest.

Suppose we want to write a DFA that recognizes strings where the third to last symbol is the same as the last symbol. We can make a DFA for this, but it's pretty atrocious:

Intuitively, this is because we want to "remember" something, and we've discussed how lack of memory can be an issue for DFAs (though in this case, it's still possible, of course). Consider the following "DFA" instead. I say "DFA" in quotes, because as you can see, it's not a DFA: there are multiple transitions per state, and some states don't have transitions for every letter.

But, if you follow the arrows, you'll notice it accepts the same strings as the first DFA. However, you can get "stuck" if you follow the wrong transition for the letter `a` or `b` from the initial state. To handle this, we'll just say that as long as there's some walk through the graph that gets you to a final state, the string is accepted, and that solves our problem. As you can see, we used far fewer states writing this automaton than the other; maybe this is a more powerful model of computation.

Let's define them more precisely.

5.1.3 NFA Definition

We call automata like this *nondeterministic finite automata*, or NFAs (maybe it should be NFAa and DFAa, because of the Greek-style pluralization...). We define NFAs to be tuples $(Q, \Sigma, \delta, s, F)$, similarly to DFAs. The only difference is in δ.

Instead of the "type" of δ being $Q \times \Sigma \to Q$, it is $Q \times \Sigma \to \mathcal{P}(Q)$, that is, the codomain is the powerset of Q. In other words, the function gives you a whole **set** of possible states you could be in! This gives us the behavior of the NFA shown above. Instead of saying something like $\delta(0, a) = 1$, we say $\delta(0, a) = \{0, 1\}$. That is, if we're in the first state, and we see the letter `a`, we can *either* go to state 0 or to state 1.

We say a string is accepted by an NFA if there is some walk that leads to a final state. To formalize this, let's define the analagous function δ^* for an NFA.

$$\delta^*(q, \epsilon) = \{q\}$$

$$\delta^*(q, x \cdot w) = \bigcup_{s \in \delta(q,x)} \delta^*(q, w)$$

Then we say that a string w is accepted if $\delta^*(s, w) \cap F \neq \emptyset$; that is, at least one of the states we can reach is a final state. Similarly to a DFA, we denote the language of all strings accepting by an NFA M by $L(M)$.

5.1.4 Remarks

We want to come up with a more powerful model of computation than DFAs, so we want to recognize languages that a DFA cannot. However, we should also make sure that our new model, NFAs, can recognize every language a DFA can! Luckily, this is quite easy to see: a DFA **is** an NFA!

To be more precise, every DFA has an equivalent NFA that recognizes the same language. If δ is the transition function for the DFA, define a new NFA with the transition function $\delta'(q, x) = \delta(q, x)\}$, and keep everything else the same. Clearly, this will recognize the same language, though we will soon prove this more formally in Coq.

Some definitions of NFAs include so-called ϵ transitions: that is, transitions that can be taken without consuming **any** input (ϵ traditionally denotes the empty string). Our definition is equivalent. While these can make writing NFAs somewhat easier, we choose not to define them that way here because they make definitions more complicated and they also cause issues for some NFA algorithms.

Furthermore, we can see that, given an NFA with ϵ transitions, we can always generate an equivalent NFA without ϵ transitions. Consider what an ϵ transition is: if we have an epsilon transition from state q to state p, then any time we can reach state q, we can also reach state p by following the ϵ transition. That is, if $q \in \delta(r, x)$ for some state r, then $p \in \delta(r, x)$ as well. Also, if we are in state q, and from state p we can reach state t by the symbol x, then we can also reach state t from state q with the symbol x. That is, if $t \in \delta(p, x)$, then $t \in \delta(q, x)$. Using these rules, we can see that there is no need to include ϵ transitions in our definitions; we can simply enlarge the sets of destination states for each state.

5.2 Coq Definition of NFAs

Let's now take the time to define these new automata in Coq. As you might expect, our basic definition is quite similar to the definition of a DFA, except we change the type of the transition function, which now returns a list. While our definition above returns a set, we can easily see this is equivalent, but it turns out to be easier to work with in Coq. We also set the types to be implicit arguments to each of the members of our structure for convenience.

```
Structure nfa (A B : Type) := {
  nt : A -> B -> list A;
  ns : A;

  nF : A -> bool;
}.

Arguments nt {A} {B}.
Arguments ns {A} {B}.
Arguments nF {A} {B}.
```

Note we'll be using some of the DFA lemmas and definitions, so you may wish to write this in the same file. Alternatively, we can import the other file by first compiling it:

```
$ coqc DFA.v
```

Then importing it in our new file:

```
Require Import DFA.
```

We can now define the equivalent of `tStar` for an NFA, which we call `ntStar`, to keep with the theme and distinguish it from the other `tStar` we already have. To do so, we'll essentially just replicate the definition given above, with one slight change to account for the fact that we're using lists instead of sets. Specifically, we use `flat_map` instead of some `union` function. `flat_map` is equivalent to mapping a function `f : A -> list B` over a list, then "flattening" out the list, hence the name. In Haskell we have:

```
flat_map = concat . map = concatMap = (=<<)
```

Other than that, it is a routine `Fixpoint` definition:

```
Fixpoint ntStar {A B : Type} (M : nfa A B) (q : A) (str : list B) : list A :=
  match str with
  | []      => [q]
  | x :: xs => flat_map (fun st => ntStar M st xs) (nt M q x)
  end.
```

Our definition of an accepted string must be slightly different to account for `ntStar` returning a list. We use the defined function `existsb`, which is the same as `any` from Haskell; essentially, it takes a function and a list, and returns true if there is any element of the list for which the function returns true. Therefore, we would have that `existsb is_even [1;2;3] = true` and `existsb [1;7;9] = false`.

```
Definition naccepted {A B : Type} (M : nfa A B) (str : list B) : bool :=
  existsb (nF M) (ntStar M (ns M) str).
```

5.3 Flat Map Proofs

We're using several library functions here, so we'll first prove some facts about them, both to learn how to do so, and secondly because we'll need these theorems to prove stuff later. This is very similar to what you'll do if you ever try to prove that some program you write (e.g., a compiler) is correct, not in scale though, of course.

We'll prove fours facts about `flat_map` that are hopefully obvious. It would be good to take the time to do the proofs yourself just for practice, however.

The first of these is that flat mapping the function which just wraps it's argument in a singleton list is the identity function.

```
Lemma flat_map_id :
  forall {A : Type} (xs : list A), flat_map (fun x => [x]) xs = xs.
```

Next we prove that flat mapping over a list formed by concatenating two other lists is the same as flat mapping over the two lists and then concatenating them. Essentially, this is a distributive-style property. Again, this is a straightforward induction proof, with a simple application of the `app_assoc` property from the Coq library which says that `a ++ (b ++ c) = (a ++ b) ++ c` for lists `a`, `b`, and `c`.

```
Lemma flat_map_app :
  forall {A B : Type} {f : A -> list B} (xs ys : list A),
    flat_map f (xs ++ ys) = flat_map f xs ++ flat_map f ys.
```

This next lemma says that there are essentially two ways to flat map two functions successively on a list. It's the analogy of the property for functors that `fmap f (fmap g x) = fmap (f . g) x`, or to specialize to lists, that `map f (map g x) = map (f . g) x`. Stated another way, we can either do two successively passes over a list with the two functions, or we can just do one pass where we apply both functions. Because of the structure of flat map, the statement of it is slightly different, however. Using the previous lemma, `flat_map_app`, this becomes a standard induction proof.

```
Lemma flat_map_comp :
  forall {A B C : Type} {f1 : A -> list B} {f2 : B -> list C} (xs : list A),
    flat_map f2 (flat_map f1 xs) = flat_map (fun x => flat_map f2 (f1 x)) xs.
```

5.4 NFA Proofs

In this section, we'll do some fairly straightforward proofs about NFAs, proving an analogy to the `tStar_step` lemma from the previous part and showing that every DFA has an equivalent NFA.

5.4.1 Step Proof

Before we had a very useful lemma called `tStar_step`, which essentially made induction possible. We want to construct some analogy of that, finding an equivalent expression for the possible states reached by strings $w \cdot x$. If we think about, then we can come up with the following statement:

$$\delta^*(q, w \cdot x) = \bigcup_{s \in \delta^*(q,w)} \delta(s, x)$$

If you recall back from a few sections ago, where we defined δ^*, this statement is **very** similar to the definition, and hopefully it makes sense. Proving this, however, turns out to be quite useful, as it converts the inductive definition from "building" on the front to "building" on the end.

The statement in Coq is as follows. Again, we replace the use of a union with `flat_map`.

```
Lemma ntStar_step {A B : Type} (M : nfa A B) :
  forall (str : list B) (x : B) (q : A),
    ntStar M q (str ++ [x]) = flat_map (fun st => nt M st x) (ntStar M q str).
Proof.
induction str.
```

As is standard by now, we proceed by induction on `str`. The base case is fairly simple, we need only note that simplifying gives us:

```
flat_map (fun st : A => [st]) (nt M q x) = nt M q x ++ []
```

in the goal. This is easily solved by a combination of the property that `xs ++ [] = xs` and the `flat_map_id` lemma from before.

```
intuition.
simpl.
rewrite app_nil_r.
apply flat_map_id.
```

The inductive case, however, ends up being a bit trickier. At first, we can proceed in a fairly standard fashion, using our `flat_map_comp` and `flat_map_equality` to get the goal to be:

```
IHstr : forall (x : B) (q : A),
        ntStar M q (str ++ [x]) =
        flat_map (fun st : A => nt M st x) (ntStar M q str)
x : B
```

```
q : A
______________________________________(1/1)
flat_map (fun st : A => ntStar M st (str ++ [x])) (nt M q a) =
flat_map (fun x0 : A => flat_map (fun st : A => nt M st x) (ntStar M x0 str))
  (nt M q a)
```

At first glance, this seems quite easy. Our inductive hypothesis tells us that for all symbols `x` and states `q`, our inductive hypothesis is true. Instinctively we may wish to rewrite "inside" the lambda in on the left hand side of the goal. If we could do this, then we would easily finish the proof. However, this is impossible in standard Coq.

Doing so is essentially equivalent to the axiom of [functional extensionality](https://ncatlab.org/nlab/show/function+extensionality), which essentially says that, quite reasonably, functions are equal if they have the same value for every input. In Coq, this might be:

```
Axiom extensionality : forall (A B : Type) (f g : A -> B), (forall (x : A), f x = g x) -> f
```

Unfortunately, while this sounds like a reasonable idea, it cannot be proved in Coq [3]. Essentially, the problem is that equality in Coq is syntactic equality: two terms are equal if they were "built" the same way. This works quite well for many types: natural numbers, lists, etc. But we can't use it to prove functions are equal because they're not, in Coq. As shown above, Coq does have support for adding new axioms: essentially, functions that we simply declare exist. In this case, adding this axiom causes no issues, because of course the functions are equal for all practical purposes.

However, we can also avoid the need for the axiom, so we prefer to do that instead. We can do this by starting *another* induction, this time on `nt M q a`. The base case is trivial, and the inductive case can be solved by a combination of the two inductive hypotheses.

```
induction (nt M q a).
intuition.
simpl.
now (rewrite IH1, IHstr).
Qed.
```

The `now` tactic (really a notation) is something new. Basically it just means "run this tactic, then try to solve it using `easy`". 'easy' is similar to `intuition` in that it can solve simple goals for us, so often this can save us a line or two at the end of proofs.

5.4.2 Building NFA for a DFA

As discussed earlier, to build an NFA that is equivalent to a DFA, we need only convert the transition function so that it returns a singleton of a state instead of a list. This can be implemented as follows:

```
Definition dfa_to_nfa {A B : Type} (M : dfa A B) : nfa A B :=
  Build_nfa A B (fun st x => [t M st x]) (s M) (F M).
```

Mirroring Proof As usual, we start by proving a "mirroring" property, essentially stating that our transition function behaves just like a DFA's transition function, with always a single destination state, except that it is wrapped in a list.

```
Lemma dfa_to_nfa_mirror {A B : Type} (M : dfa A B) :
  forall (str : list B), [tStar M (s M) str] = ntStar (dfa_to_nfa M) (ns (dfa_to_nfa M)) str
Proof.
apply rev_ind.
intuition.
```

We use `rev_ind` for "reverse" induction: building the string up by appending, rather than prepending, as usual. The base case is trivial. The inductive case's goal is:

```
H : [tStar M (s M) l] = ntStar (dfa_to_nfa M) (ns (dfa_to_nfa M)) l
______________________________________(1/1)
[tStar M (s M) (l ++ [x])] =
ntStar (dfa_to_nfa M) (ns (dfa_to_nfa M)) (l ++ [x])
```

We can apply both `tStar_step` and `ntStar_step` to get the goal into a form that works with our inductive hypothesis. Finally, we can solve it with the hypothesis, `H`. Below I've combine all these rewrites for brevity.

```
intuition.
now (rewrite tStar_step, ntStar_step, <- H).
Qed.
```

Correctness Proof Now we can prove correctness of this NFA: it accepts precisely the strings that the original DFA did. In Coq, we write:

```
Theorem dfa_to_nfa_correct :
  forall {A B : Type} (M : dfa A B) (str : list B),
    accepted M str = true <-> naccepted (dfa_to_nfa M) str = true.
Proof.
```

Both directions are fairly similar, so I'll only do the forward direction here. We wish to show:

```
H : accepted M str = true
______________________________________(1/2)
naccepted (dfa_to_nfa M) str = true
```

Unfolding the definitions of `accepted` and `naccepted` make this far less opaque:

```
H : F M (tStar M (s M) str) = true
______________________________________(1/2)
existsb (nF (dfa_to_nfa M)) (ntStar (dfa_to_nfa M) (ns (dfa_to_nfa M)) str) =
true
```

In fact, by using our `dfa_to_nfa_mirror` property, we can solve this quite easily, because we know the list we are checking with `existsb` is a singleton.

```
unfold accepted in H.
unfold naccepted.
rewrite <- dfa_to_nfa_mirror.
simpl.
now (rewrite H).
```

5.5 Equivalency Proof

Now we should discuss our original objective with defining NFAs: to create a more powerful model of computation. It may seem like we have: DFAs are definitely not more powerful than NFAs, and NFAs can often produce smaller automata.

However, it is not the case. In fact, every NFA also has an equivalent DFA: that is, the languages that NFAs can recognize are exactly the same as the languages that DFAs can recognize.

How do we see this? Constructively, of course! Given an NFA, how do we create an DFA that recognizes the same language?

We use what is called the *powerset construction*. Essentially, our states will be sets of states, rather than "single values", as we have shown before.

Let's start defining the DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$, given an NFA $N = (Q, \Sigma, \delta, s, F)$. Starting with the easiest parts:

The alphabet is the same, as expected. The start state is just the singleton containing the original start state: $s_M = \{s\}$. As I stated, our states are sets of states. To be safe, we'll throw every single possible set of states in (i.e., the powerset, hence the name), though this is often unnecessary (we'll discuss DFA minimization later). So $Q_M = \mathcal{P}(Q)$. Recalling that our states are sets, our final states are those which containg at least one final state from the original NFA:

$$F_M := \{\mathcal{S} \in Q_M : \mathcal{S} \cap F \neq \emptyset\}$$

Finally, the transition function is as follows: $\delta_M(q, x) = \bigcup_{s \in q} \delta(s, x)$

Note that, even though the output of this function is a set of states, this is still a DFA, because our states are themselves sets of states.

Definitions Let's now prove that this really works.

The first step, as always, is to define everything in Coq. The definitions follow the above, so I won't say too much, except that to note that we again use

lists instead of sets, and use `flat_map` in place of unioning. Note the similarilty between these definitions and the definitions we gave earlier of an NFA and what it means to be accepted.

```
Definition powerset_nfa_trans {A B : Type} (M : nfa A B) (possible : list A) (x : B) : list
    flat_map (fun source => nt M source x) possible.

Definition powerset_nfa_f {A B : Type} (M : nfa A B) (possible : list A) : bool :=
    existsb (nF M) possible.

Definition powerset_nfa {A B : Type} (M : nfa A B) : dfa (list A) B :=
    Build_dfa (list A) B (powerset_nfa_trans M) [ns M] (powerset_nfa_f M).
```

Mirroring Proof We can now prove the version of mirroring property that we will need for this construction, which is:

$\delta^*_M(q, w) = \delta^*(q, w)$

In Coq, this becomes:

```
Lemma powerset_nfa_mirror {A B : Type} (M : nfa A B) :
    forall (str : list B),
      tStar (powerset_nfa M) (s (powerset_nfa M)) str = ntStar M (ns M) str.
Proof.
apply rev_ind.
```

There is one particularly interesting thing about this statement, which is that it's actually much stronger than the mathematical statement we wrote. Specifically, for the lists to be equal, they must not only have the same elements, but also in the same order. Luckily for us, it's still true, and this is the easier way to prove things when using lists.

We proceed by `rev_ind` as shown above, and the base case is trivial. After using `tStar_step` and `ntStar_step`, we end up with:

```
H : tStar (powerset_nfa M) (s (powerset_nfa M)) l = ntStar M (ns M) l
______________________________________(1/1)
powerset_nfa_trans M (tStar (powerset_nfa M) [ns M] l) x =
flat_map (fun st : A => nt M st x) (ntStar M (ns M) l)
```

Though it's not immediately obvious how to prove this, by unfolding the definition of `powerset_nfa_trans`, it becomes clear. The proof is below:

```
intuition. (* For the base case *)

intuition.
rewrite tStar_step, ntStar_step.
simpl.
unfold powerset_nfa_trans.
now (rewrite <- H).
Qed.
```

Correctness Proof The correctness proof is similar to those we've done before, so I'll only do the first direction, as before.

The statement of the theorem is:

```
Theorem powerset_nfa_correct :
  forall {A B : Type} (M : nfa A B) (str : list B),
    accepted (powerset_nfa M) str = true <-> naccepted M str = true.
Proof.
intuition.
```

By unfolding the definitions of `accepted` and `naccepted`, we get:

```
H : powerset_nfa_f M (tStar (powerset_nfa M) [ns M] str) = true
______________________________________(1/2)
existsb (nF M) (ntStar M (ns M) str) = true
```

Using the mirroring property gives:

```
H : powerset_nfa_f M (ntStar M (ns M) str) = true
______________________________________(1/2)
existsb (nF M) (ntStar M (ns M) str) = true
```

Now that these two things actually have the same definition, so Coq can finish this for us if we use `now`. The proof of the forward direction is therefore:

```
unfold accepted in H.
simpl in H.
unfold naccepted.
now (rewrite (powerset_nfa_mirror M) in H).
```

5.6 Conclusion

We have defined a new model of computation, called an NFA, which is equivalent to DFAs. However, they still have their uses. Though we won't prove this, NFAs can be exponentially smaller than the equivalent DFA. Intuitively, this should make sense: the powerset construction gives us a DFA with 2^n states, where the original NFA had n states.

Not only this, but in practice it's often easier to come up with an NFA that recognizes a given language than to come up with a DFA. Then we can simply invoke the powerset construction to get a DFA. Though we have yet to talk about performance, the current methods for things are somewhat inefficient, especially if we just keep building larger and larger DFAs. Later, we'll talk about minimizing DFAs to make some of this more computationally feasible.

In the next couple parts, we'll talk about proof automation, to make the proof writing process less tedious, and then we'll discuss regular expressions and their relation to NFAs and DFAs.

References

[1] Curry-Howard correspondence. https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence.

[2] Intuitionistic Logic. https://en.wikipedia.org/wiki/Intuitionistic_logic.

[3] The Logic of Coq. https://github.com/coq/coq/wiki/The-Logic-of-Coq#axioms.