All Posts
A Formal Introduction to Models of Computation in Coq (Part 2)
A Formal Introduction to Models of Computation in Coq (Part 1)
Why Prolog?

A Formal Introduction to Models of Computation in Coq (Part 1)

Introduction

This series will work through some introductory material on models of computation (based on this course. In particular, this article follows this). There are two goals: to teach the material itself, as well as teach how to use Coq/the basic theory behind it.

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.

Introduction to Coq

Coq is a system for theorem proving, called a *proof assistant*. It includes several sub-programming languages for variable specifications of definitions and proofs.

Curry-Howard Isomorphism

The whole thing is based on the Curry-Howard isomorphism; 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:

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

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.
```

Definining Strings

We will define strings inductively, because it's the most natural way to do it. We're 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.
```

Defining Basic 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.

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.

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).
```

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.

Proving Basic String Properties

Basic Lemmas

In this section, we will prove the following lemmas, which will be useful to us throughout the rest of this article.

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.

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.

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!).

Various Lemmas

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

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.
```

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.

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.
```

Conclusion

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 a little more work).

This series will continue soon, going more deeply into strings and languages, followed by automata and more. The following parts will also be visible on this blog.

Files

All files used in this article can be downloaded here.

Older Ne

All Posts
A Formal Introduction to Models of Computation in Coq (Part 2)
A Formal Introduction to Models of Computation in Coq (Part 1)
Why Prolog?

A Formal Introduction to Models of Computation in Coq (Part 2)

Introduction

Note all the full text for all proofs are available at the GitHub repository.

Last time 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.

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 importan and will be addressed shortly. First, however, we will try to give some intuition into wh 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 `mod` 2 == 0
```

This definition is concise and familiar to any programmer, but it requires the definitior of several more terms, and is therefore harder to prove theorems about. Therefore, it 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 tw 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" 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.

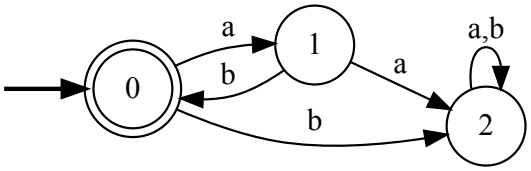
Now we have some intuitive sense of what computation is, but no sense of what the "rules" are, so to speak. Clearly there are some things we can compute, and there are things we can't compute. For example, as you may know, most real numbers are uncomputable. But without defining precisely what a computation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

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, Σ, δ, 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.

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, the 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.

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 transition, 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. Co has support for functions, so we'll have a function t : A -> B -> A (the curried version of : 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}.
```

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 the inside the curly brackets, {}.

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

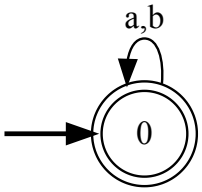
```
Definition accepted {A B : Type} (M : dfa A B) (str : list B) : bool :=
  F M (tStar M (s M) str).
```

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

What languages can be recognized by a DFA?

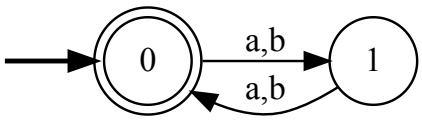
This is a complicated question, but we can say some things with little effort.

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, $L_1 \cap L_2$? 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.

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."

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 state 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 $\overline{M}$ w are also in state q. This lemma makes proving the above theorem quite simple. We car 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 c are different), as follows:

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

Note all the full text for all proofs are available at the GitHub repository.

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 $\delta^*(s, w) = \bar{\delta}^*(s, w)$, then for any $x \in \Sigma$, we have $\delta^*(s, x \cdot w) = \bar{\delta}^*(s, x \cdot w)$. Of course, this is true, but we have no information abo what state we end up in after processing x! Whereas, we do know something about th

state $\delta^*(s, w)$. We want to show the following instead: if $\delta^*(s, w) = \overline{\delta}^*(s, w)$, the 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 beer 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: onc 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 ho 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, becaus we essentially want to prove that the final function was defined correctly. To do so, w 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, w 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.

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, w 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 DFA $F_1 \times F_2$.

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 keep 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)) str)
```

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 pai 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 ca 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, an 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 o 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)) str)
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 "Fs" 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 $L_1 \cap L$

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 $L_1 \backslash \text{up} L_2$ 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.

Conclusion

We have shown some fairly general properties languages which are recognizable by DFAs. In the next part, we will expand on this, showing that we may express first order logic predicates in terms of automata, and therefore that we may algorithmically decide the truth value of any formula expressed in this manner.

Note all the full text for all proofs are available at the GitHub repository.

All images generated by this website.

Older