Introduction

Okay, so I am going to be talking a bit about formal verification. There are a few primary goals of this talk:

- Explain why you should be interested.
- Very briefly cover some of the methods. We're mostly going to focus on theorem proving, and dependent types.
- Make proof assistants more accessible. One of the things that I had problems with was understanding just what this thing was. It's not at all clear how any of the pieces fit together unless you find the right resources.
- Give some rough intuitions about how systems like these work. They're not magic, and in fact if they were that would be a problem because magic can't be trusted.

Preface

Before we begin I'd like to make sure we're all on the same page with respect to a few things.

- Type signatures:

In this talk I'll be using Haskell-style type signatures, so here's a brief introduction. If "x" is an identifier, then

```
x :: A
```

Means that "x" has the type A. Note that while Haskell uses two colons, one colon is also standard. This will be mixed a bit in the talk, but it should be clear from context. Note that "A" can be any type we want.

```
x :: Integer
y :: [Integer]
```

Would mean that x is an integer, and y is a list of integers.

In languages like these pretty much everything has a type. Even functions. For instance we might say that addition has the type:

```
(+) :: Integer -> Integer -> Integer
```

This may seem peculiar for some of you, but it literally just means that addition takes these two integers as an argument and produces an integer result. The arrows are a common notation in type theory, and there is good reason for it as we shall see.

Also note that a type may be polymorphic. For instance you can have something like

```
id :: a -> a
```

What this means is that `id` is a function which takes an argument of some type `a`, which could be `Integer`, `String`, or anything you can possibly imagine, and returns a value of that same type. In this case there is only one function which matches this type signature, and that's the identity function which returns its argument.

```
id :: a -> a
id x = x
```

- Basic lambda calculus:

We'll also need to talk a tiny bit about the lambda calculus. More or less you just need to know that lambda calculus consists of lambda terms which can be

- variables
- `(\x . t)` where '`t`' is another lambda term (lambda abstraction)
- `(ts)` (application)

Aside from that you just need to know one thing: lambda calculus is just substitution. For example the id function could be written in lambda calculus like

```
(\x . x)
```

No matter what this is applied to we get the same thing back.

```
(\x . x)t
t  -- Substituting the x for the t.
```

Formal Verification: What it be?

Alright! So, what is formal verification? Tautologically speaking formal verification is the use of formal methods to verify properties of programs. In short "proving programs correct".

Formal verification is inherently appealing. You want things to be correct, and work as intended. Bugs might not matter too much depending on the application, after all it's not the end of the world if your whoopy-cushion app crashes every so often, but maybe it is if an ICBM has an unexpected integer overflow. In many cases bugs can result in serious harm, death, and large expenses.

In this day and age pretty much every device has some form of computer in it. If you have ever written a program, or even used any software this should be a terrifying fact! Really you should be afraid to use elevators because it might run Java and somebody doesn't like to check their damn null references. So formal verification is important for these situations. We can't afford for some programs to fail.

A related goal is actually that mathematicians want computers to formally verify proofs as well, particularly since any given theory or paper now consists of pages upon pages worth of proofs which are extremely difficult to check by hand. It'd be nice to have a computer check our logic.

Formal verification can be pursued in a number of ways:

- Manual labour (Kidnap mathematicians and make them do your dirty work – expensive, and error prone)
- Model checking (essentially boils down to checking every possible state of your program and whether or not it adheres to a given property – proof by exhaustion)
- Type checking (in its roughest sense types are supposed to provide guarantees about how values in a program behave). This can be really good (Haskell), or done incredibly poorly, like with Java and Python. For instance in Haskell you know that a function which takes an int and returns an int probably isn't going to explode or do anything unexpected, however in Java you can get unexpected null references, and in Python you can get pretty much anything. Unpleasantness all around.
- Theorem proving (Math proofs on your program for great justice. can actually be viewed as extended type checking, as we shall see)

Formal verification can be done at many, and even multiple levels of abstraction:

- High level: Are the algorithms correct? Does this program implement the algorithm correctly?
- Low level: The aerospace industry actually meticulously checks over machine code spit out by their compilers. It's what's run, so they need to

make sure it's squeeky clean. Otherwise we risk planes falling out of the sky.
- Hardware: Even the design of hardware itself undergoes verification sometimes. For instance Intel is pretty big on this ever since their Pentium floating point division bug cost them millions.
- And of course you can mix and match all of these levels.

We're going to focus on higher levels of abstraction for the most part. Proving that your algorithms are correct, and assuming that the underlying systems are working fine. Experience tells us they're probably not, but hey! It's better than nothing. The proof assistant we'll be using does satisfy the de Bruijn criterion, however. I.e., all of the parts involved in verifying the correctness of proofs are small and simple. We'll see this shortly.

Coq

We're going to look at some examples of a proof assistant in action. The one we'll be using is one of the staples called Coq. So named because of:

- Tradition of naming programming languages after animals (OCaml).
- French. Coq had its start in France. Words are different there, I am told
- CoC: Calculus of Constructions. The theory upon which Coq is based
- Thierry Coquand. Creator of the Calculus of Constructions
- The universe is conspiring against us here!

Examples

We're now going to go over some quick examples because if you just look at the code you're going to be very confused at first.

If you're familiar with functional programming in the ML dialects some of this should be fairly familiar to you. This is not terribly different than how you would write a program in Haskell or Ocaml. It's a similar functional style. That said you should be aware that Coq is not Turing complete (as we will discuss later), and as a result you will not be able to express certain programs within it. All Coq programs must terminate. By default all recursion must be structural (think recursing on smaller and smaller subsets of a data structure), however as long as you can show that a recursion is well-founded then you should be able to implement it in Coq, but you will have to prove it.

Beyond the typical functional programming, Coq supports a tactics style of proof. This is what looks weird and confusing until you actually walk through an example. What it boils down to is writing down the steps for a math proof. However, in math proofs you often rewrite things explicitly, constantly reminding the reader of the current state of the proof. The proofs in Coq consist of brief tactics which don't include the proof state, and it's pretty much impossible to read at first – you really need to step through the proofs in Coq because it will show you the proof state.

We'll be using ProofGeneral, which is an emacs mode for interacting with proof assistance. It has three main buffers. The file you're working with, the current goals and the proof state, and any other responses from the proof assistant.

First let's prove that addition of natural numbers is associative.

```
Theorem plus_associative : forall (a b c : nat),
                             a + (b + c) = (a + b) + c.
Proof.
  intros a b c. (* Introduce a b and c. *)
  (* Perform induction on a. This gives us two goals. One for the base case, and one
  for the induction hypothesis. *)
  induction a.

  simpl.        (* This is the base case. We should be able to simplify this expression some
  reflexivity.  (* This got rid of the 0 term, so we can finish the goal with reflexivity. *

  (* Now we're in the second case. Need to show that given the induction
     hypothesis it's true for the successor to a *)
  (* Again we can try to simplify. This does some basic evaluation, and
     tells us that S a + n is the same as S (a + n) *)
  simpl.

  (* Notice that these aren't quite the same yet. The associativity is different.
     We can use the induction hypothesis to rewrite one side *)
  rewrite IHa.
  reflexivity.  (* And again, reflexivity finishes the goal *)
Qed.
```

Being presented with just this code is like a fire-hose. It's much easier to stomach when you can look at the proof state in the proof assistant!

Let's look at something a bit more complicated. Let's show that the sum of natural numbers up to n is equal to n * (n + 1) / 2. We'll need to import some things right now. Don't worry about it.

```
Require Import List.
Require Import Coq.Arith.Arith_base.
Import ListNotations.
Require Import NAxioms NSub NZDiv.
Require Import Coq.Numbers.Natural.Abstract.NDiv.
Require Import Coq.Numbers.Natural.Peano.NPeano.
```

```
(* Definition for a sum from 1..n *)
Definition natSum (n : nat) : nat :=
  (n * (n + 1)) / 2.

(* We could also write this recursively, but this is obviously less efficient. *)
Fixpoint regularSum (n : nat) : nat :=
  match n with
    | 0 => 0
    | S n' => n + regularSum n' (* Match the natural number with a successor of a nat *)
  end.

(* Note that these compute the same values... *)
Eval compute in natSum 10.
Eval compute in regularSum 10.

(* So, maybe we should show that these two definitions are actually equivalent! *)

(* First let's start with a lemma. Sum from 1 to S n (successor, or n + 1) is the same as
   The S n plus the sum from 1 to n.
*)

Lemma natSum_S : forall (n : nat),
                     natSum (S n) = S n + natSum n.
Proof.
  intros n.  (* Introduce our variable n *)
  unfold natSum.  (* replace natSum with its definition *)

  (* We'll change all of the successors to addition with 1 *)
  (* Replacement will gives us an extra goal, because we have to prove that S n = n + 1 late
  replace (S n) with (n + 1).

  (* Move the addition on the right hand side into the division. This requires
     that the denomenator is non-zero, so that's added as a goal as well.
  *)
  rewrite <- Nat.div_add_l.

  (* Now we replace the numerator to make them equal. We have to show that this
     replacement is valid too, so another goal is added.
  *)
  replace ((n + 1) * (n + 1 + 1)) with ((n + 1) * 2 + (n * (n + 1))).
  reflexivity. (* Reflexivity finishes the current goal. *)

  (* We will use associativity, and simplification to show that n + 1 + 1 = n + 2 *)
  rewrite <- plus_assoc.
  simpl.
  (* Commute one particular term -- adds a goal *)
  replace (n * (n + 1)) with ((n + 1) * n).
  rewrite <- mult_plus_distr_l.  (* Use distribution principle *)
  replace (2 + n) with (n + 2).  (* Commutativity of addition of a specific term *)
  reflexivity.  (* Now both sides are equal, so we can use reflexivity to finish *)

  (* This is solved with commutativity of addition *)
  apply plus_comm.

  (* This is just commutativity of multiplication *)
  apply mult_comm.

  (* Coq can automatically solve this *)
  auto.

  (* And this is an existing theorem too! *)
  apply Nat.add_1_r.
Qed.


(* Now we can try to show that natSum is the same as regularSum! *)
Theorem regularSum__natSum : forall (n : nat),
  natSum n = regularSum n.
Proof.
  induction n.  (* We'll show this by induction on n *)

  compute.  (* Base case can just be computed *)
  reflexivity. (* Reflexivity finishes the current goal *)

  (* Note that we now have an induction hypothesis. *)
  unfold regularSum. (* We can unfold regularSum *)
  fold regularSum. (* This took out the S n term, we can fold the rest again *)

  (* Now using our induction hypothesis we can rewrite this to be exactly our lemma! *)
  rewrite <- IHn.
  apply natSum_S.
Qed.
```

Continuing On...

So, how does Coq work? It actually relies entirely upon type checking. You might think that's insane, after all what does type checking have to do with proving theorems? As it turns out just about everything due to the Curry-Howard isomorphism.

If you're a functional programmer you have probably heard of this before. "Programs are proofs", but what does this mean, and how is this the case?

It's actually a pretty simple idea. Types are propositions, and any program of that type serves as an "existence" proof of that proposition. We'll see an example of how this works in a second. For the purposes of this presentation we will just cover the original Curry-Howard isomorphism, which relates the simply typed lambda calculus and intuitionistic logic. Coq uses something similar, but it actually uses a different typed lambda calculus under the hood called the calculus of inductive constructions, which gives it a bit more power because it has quantifiers and such built in. The simply typed lambda calculus can give you the right intuition, anyway. In this system

- Any type which is inhabited (has a value) represents a provable proposition.
- We represent implication with "->" in types
- Conjunction is a tuple (a, b)
- Disjunction is an Either type. (Either a b)
- False is represented by an uninhabited type. We call this type Void.
- Negation is a -> Void

Of course we have issues with general recursion, so Coq actually requires that all programs terminate. To see why this is an issue, consider if you had a function which loops forever. The type this function returns could be any type whatsoever, which would mean all types are inhabited, so everything becomes inconsistent and horrible. Once you have False = True your system becomes entirely useless.

Let's look at an example of this. Let's say we want to prove `A -> B -> A`. If you were to provide a function with this type, note that `A` and `B` can be ANY type, what must this function be?

```
??? :: a -> b -> a
```

Well, you might say this type is inhabited by the function `const`, which always returns the first argument.

```
const :: a -> b -> a
const a b = a
```

Looking at this it makes intuitive sense that this would form a proof of the proposition. Both of the arguments to the function are propositions, and by returning the first proposition, `a`, we show that `b -> a`, since regardless of whether `b` is true or false we can produce `a` when given `a`. Explaining it is confusing, but it actually makes a lot of sense. If you're given `a` and `b`, then of course you can prove the proposition `a` from that information.

And of course you can write this as a simple lambda term

```
(\x : a, y : b. x)
```

As we know the lambda calculi are very simple, essentially consisting of one axiom (beta reduction), which just acts like replacement. The typed lambda calculi add a bit more pizzazz, but they're pretty much the same. This is how Coq can allow for complicated proof tactics, while still satisfying the de Bruijn criterion. The tactics are merely a means of manipulating lambda terms in an automated and useful fashion. It doesn't matter what the tactics do, even if they're technically "incorrect", because pretty much all Coq does is type checks the generated lambda term. If this term is correct, then your theorem is proven, and otherwise it is not. How this term was generated is completely irrelevant.

How might you use this / wrapping up

Now after some learning you can learn this and start proving things about your programs. Coq provides a means for extracting the code into other programming languages, so you can write some pure functions in Coq, and prove properties on them, and then extract them into Haskell, or Ocaml code for use with anything else. It's a bit complicated to pick up, but it can definitely help with mathematical proofs and more as well. While it may be a large burden to fully verify every portion of your program, Coq provides vital tools for proof automation which can significantly narrow down this effort (I did not show this, but Coq provides Ltac which allows you to write your own tactics. You can have very powerful tactics which do a lot of the heavy lifting for you). Further more you can just use it on small portions of your code. You don't have to verify everything, and it can still be useful to have a core piece of your software verified. Beyond that it's useful to just have your code in such a system that can help you reason about it.