Dependent Types:
the Theorem-Proving Perspective

Lawton Nichols

March 18, 2015

1 Introduction

Shall I be dependent or independent?

Most modern programming languages draw a firm line between types and expressions. "Independent" types are types that prefer the company of other types—the polymorphic type **List** A, for example, might take the type **Nat** as an argument and produce the type of lists of natural numbers, which is yet another type; that same type constructor, though, would have firmly rejected any expression that came close to it. Passing along 0 to the **List** A type constructor will not give back the type of lists that contain only 0, and instead the programmer will receive a stern talking-to from the type checker, who doesn't have time for these kinds of games. But sometimes it may be valuable to have a type like **List 0**—indeed, if checked properly, it would guarantee that there would never be any surprise 1s popping up inside a term of this type, which is a very powerful idea. Our story begins here.

1.1 You Got Your Terms in My Types!

Dependent types are the Civil Rights Act for programming languages. They break down the artificial barrier between terms and types, and allow for types to reference arbitrary expressions involving terms. **List 0** would have to be rewritten as **List Nat 0** for paradox-preventing purposes, but this type is perfectly representable with dependent types. Since it's more than just a **List**, let's call the type **ListOfOnly**, and see how we might define it.

The type **ListOfOnly** A x could be represented as the type

$$\Pi A{:}\mathbf{Type}\,.\,A \to \mathbf{Type}$$

(don't worry about the Π binding just yet—for now think of it as a λ that is creating a function on **Type**s). This type expression takes an A, which is a **Type**; a value of that type (the second A that's all by itself); and it returns back a new type. We can then define this whole expression to be **ListOfOnly**. Next we need ways to create terms of this type. The constructor for **nil** should take values for A and x before returning a **ListOfOnly** A x; and **cons** should take values for A and x again, a new value y of type A (the head of the new list), *a proof that* $y = x$, and a **ListOfOnly** A x (the tail) before returning another **ListOfOnly** A x. The reader is encouraged to think about why each of these arguments are necessary.

1.2 The Canonical Example, Just to Get It out of the Way

There is a "Hello, world!" dependently-typed program, and if we do not go over it the dependent type police will come to the author's door and relieve him of his Π privileges. To show it, we must first pick a dependently-typed language. There are several to choose from, with Agda [1] and Idris [9] being the most widespread all-purpose dependently-typed programming languages. This paper covers the powerful theorem-proving side of dependent types, and so we will use the language Coq [3], arguably the current most popular theorem-proving language with dependent types. With that, let's proceed to the example.

Much like we can represent lists of a given type that hold only one particular value of that type, we can also represent lists of particular sizes at the type level. A quirk of history has attached the name **Vector** A n to this type. Here is the start of the definition of this new type in Coq:

```
Inductive Vector (A : Type) : nat -> Type :=
```

Since Coq is a theorem-proving language, it is necessary for all theorems (and therefore expressions) representable in the language to terminate, because otherwise a proof of any proposition could be achieved with an expression that never terminates. Thus Coq does not allow recursive data types (which can be built from the top down) and only allows inductive data types (which are built from the bottom up) to be defined. This definition is for an inductively defined type called **Vector**, which takes a **Type** called A that must always be the same inside the definition, a **nat** that is allowed to vary inside the definition, and then it finally returns a new **Type**. This variability is the difference between the things that come before and after the main ":". There are two ways to make a **Vector**:

```
| vnil  : Vector A 0
```

It could be **vnil**, which means it must have length 0, or it could be of nonzero length:

```
| vcons : forall (n : nat), A -> Vector A n -> Vector A (S n).
```

If the **Vector** is built using **vcons**, it takes (excluding the argument `n` of type `nat`) the new head element, a **Vector** of any size (the reason behind the forall), and it returns a **Vector** of length *one more* than the previous one. Due to the way we defined this type, whenever we make a **Vector** it will always have its size in its type, and this size will always be the correct size. Notice that, with this definition, there is no possible way to construct a **Vector** with an incorrect size.

Dependent types are so powerful because of their support for the $\forall$ quantifier, which can create a type inhabited by a potentially infinite number of objects. For example, thanks to dependent types we can easily encode the **vcons** constructor once and have it work for an infinite number of different values of ***n***—the ***n*** is simply required as an extra argument to the **vcons** constructor. We will cover this proof/type/set correspondence in the next section.

1.3 Here Be Dragons

The next section will jump between several different yet equivalent notations. All of the following are equivalent in type theory:

- x has type A (x:A)
- x is an element of the set A ($x \in A$)
- x is a proof of the proposition A

Each of these perspectives will be used at least once.

1.4 The Rest of This Paper

Section 2 covers the theory behind dependent types and why it is so powerful. Section 3 contains a case study that uses Coq to define and prove something interesting about the **ListOfOnly** type. The Appendices provide full listings of all the Coq code mentioned.

2 The Theory Behind It All

Coq is based on a flavor of dependent types called the Calculus of Inductive Constructions [5, 6]. It is powerful enough to embed higher-order logic, which is perfect for proving theorems. Often we want to be sure that a certain proposition is always true, and so having the universal quantifier at our disposal is quite beneficial; we've already had to use it in our **Vector** example. But this is a programming language—how do we get logic inside of it? Thanks to the Curry-Howard correspondence, it was already there to begin with.

2.1 $\forall$'s Well That Ends Well

How does one create a program that represents a proof involving the universal quantifier? Martin-Löf might respond, "A proof of $(\forall x \in A)B(x)$ is a function . . . which to each element a in the set A gives a proof of $B(a)$" [10]. For every a there is a different proof, and each proof is necessary to prove the $\forall$. Since programs can be thought of as sets as well as proofs, we can see that we have more than one set to work with whenever a $\forall$ quantifier is used. This suggests that the set representing $(\forall x \in A)B(x)$ is a product of sets, where each smaller set contains a proof $B(x)$ for that particular x to which it is tied. This would look something like $(\Pi_{x \in A})B(x)$, which is closer to the syntax we saw earlier.

Π is $\forall$, and $\forall$ is Π. The proof of $B(x)$ clearly can involve x, and we might say that the result of $B(x)$ *depends on* the value of x. Now we've come full circle. If we make Π a type constructor, and let the set A range over expressions as well as types, we can represent logical proofs involving the universal quantifier. All we need now are ways to create and use those proofs.

Figure 1 shows one system of describing and using Π types, introduced by Harper [7]. The Π-FORMATION rule describes when a Π type is

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x{:}A \vdash B_x \text{ type}}{\Gamma \vdash \Pi x{:}A.B_x \text{ type}}\ \Pi\text{-}\textsc{formation}$$

$$\frac{\Gamma, x{:}A \vdash M_x : B_x}{\Gamma \vdash \lambda x{:}A.M_x : \Pi x{:}A.B_x}\ \Pi\text{-}\textsc{introduction}$$

$$\frac{\Gamma \vdash M : \Pi x{:}A.B_x \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : [N/x]B}\ \Pi\text{-}\textsc{elimination}$$

Figure 1: Π rules

well-formed, given a judgement $\Gamma \vdash \tau$ type that says τ is a well-formed type under assumptions Γ. The subscripted x in B_x emphasizes that x may occur in B.

The λ operator no longer introduces an ordinary function, but instead a dependent function type. Observe that if x does not appear in B_x, then we have a function type that does not depend on its argument. We still have a proof of $\forall x{:}A.B_x$, but there will only be one set generated by this type; this therefore corresponds to the normal function type ($A \to B$). Only when x does occur in B_x do we have the true dependent function type that represents a proof of $\forall x{:}A.B_x$.

Using (or eliminating) a dependent function is the same as using an "independent" one, except the return type now may vary with the input of the function.

2.2 A MidΣmer Night's Dream

We can now represent $\forall$, but how about its partner in crime $\exists$? What does it mean to have a program/proof of the form $\exists x{:}\tau.B_x$? If you wanted to prove to me that there exists some x such that property B_x is true, I wouldn't believe you until you showed me a specific x together with a proof of B_x that uses it. You'd have to show me that *pair* of things before I trusted you.

Similar to $\forall$, $\exists$ can also be thought of as the disjoint union (or *sum*) of a family of sets. $(\exists x \in A)B(x)$ could be represented as a potentially infinite list of $B(x_1) \vee B(x_2) \vee B(x_3) \vee \ldots$ for each possible x_i in A, representing the idea that *at least one* of the disjuncts is true. Just as Π represents several conjunctions, we use Σ to represents several disjunctions that range over a set/proposition/type. Σ is $\exists$, and $\exists$ is Σ. Σ types are also referred to as "dependent sum types".

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x{:}A \vdash B_x \text{ type}}{\Gamma \vdash \Sigma x{:}A.B_x \text{ type}}\ \Sigma\text{-}\textsc{formation}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : [M/x]B_x}{\Gamma \vdash \langle M, N \rangle : \Sigma x{:}A.B_x}\ \Sigma\text{-}\textsc{introduction}$$

$$\frac{\Gamma \vdash M : \Sigma x{:}A.B_x}{\Gamma \vdash \mathsf{fst}(M) : A}\ \Sigma\text{-}\textsc{elimination}_l$$

$$\frac{\Gamma \vdash M : \Sigma x{:}A.B_x}{\Gamma \vdash \mathsf{snd}(M) : [\mathsf{fst}(M)/x]B_x}\ \Sigma\text{-}\textsc{elimination}_r$$

Figure 2: Σ rules

Σ types are sometimes called dependent pairs, too, and Figure 2 shows the reason. The rules are once more taken from Harper [7].

There are again rules for the formation and introduction of a Σ type, and the previous paragraphs explain why they look the way they do. There are now two ways to eliminate a Σ type, because it is possible that the programmer may want to extract the value or the proof of the given property using that value. Again notice that if x does not appear in B_x, the type $\Sigma x{:}A.B_x$ behaves just like the ordinary product type $A \times B$, since B_x does not vary with x in that case.

2.3 What to Take Away

Just as Π types are a more powerful version of the $\rightarrow$ type, Σ types are a more powerful version of the $\times$ type, and the weaker types are still perfectly representable in their stronger counterparts. The typing rules show just how the types may change with the values with which they are constructed. Coupled with the ability to inductively define dependent data types, we now have the tools we need to construct a term with a type like Πx:**Nat**.x is even $\vee$ x is odd, which would correspond to a *proof* of this *theorem*. Here comes the rooster.

3 Case Study: Coq, Dependent Types, and Proving Theorems

Before jumping in and creating our **ListOfOnly** ***A x*** type, let's first see what we can do with the **Vector** ***A n*** type we've already defined in Coq.

Now is the time to install Coq if you haven't already! Towards the end of this section there will be simply too much output to copy over. All the Coq code is available in order in the Appendices, and it will be assumed henceforth that the reader is stepping through with the text and observing the output.

3.1 Using Definition and Check

Let's type the following line into Coq:

```
Definition one_two_three :=
  vcons nat 2 1 (vcons nat 1 2 (vcons nat 0 3 (vnil nat))).
```

This creates a **Vector nat 3** called `one_two_three`, representing the list [1, 2, 3]. A `vcons` requires as arguments values for `A` (from the top-level definition), a `nat` called `n` (the size of the tail, from the `forall` in the `vcons` definition), something of type `A` (the new head), and something of type `Vector A n` (the new tail). Similarly, a `vnil` requires only a value for `A`.

We can then ask Coq to type check this definition for us and confirm that we were thinking of the correct type:

```
  Check one_two_three.
```

To which Coq responds:
one_two_three
: Vector nat 3
We were right! But that was a lot of effort typing in the sizes and types when Coq could have done that for us. Let's try again:

```
Definition one_two_three2 :=
  vcons _ _ 1 (vcons _ _ 2 (vcons _ _ 3 (vnil _))).
Check one_two_three2.
```

And again we get back:
one_two_three2
: Vector nat 3
The _ allows us to have Coq fill in the information when it can very easily get it from somewhere else; even the size can be omitted and automatically deduced.

With this useful ability to define and type check things, let's move on and define the **ListOfOnly** type.

3.2 Defining ListOfOnly

The type **ListOfOnly** itself must contain a **Type** called A (the type of objects allowed in the list) and a value x of type A (the value of the sole item that is allowed in the list). The way to start the definition of this type would therefore be:

```
Inductive list_of_only (A : Type) (x : A) : Type :=
```

After both of those arguments are provided the result is a new **Type**[1].

There are two types of constructors for this list, just like every other list: **nil** and **cons**. **nil** does not require anything other than values for **A** and **x**, and these values are already implicitly required, so the constructor can be implemented as so:

```
  | mynil  : list_of_only A x
```

cons requires an actual element to go in the list (the head) and another list (the tail). But this isn't a normal list—we need more than just those things! This is a list *of only* xs, and so we need to be sure of that nothing else can appear in the list. In addition to the new head of the list, y, we need a proof that y is the same as x; otherwise this list could contain any element. So, *for every* possible value of y, we can build up a new list only if we are given a proof that $y = x$, written in Coq simply as `y = x`. The constructor must therefore look like:

```
  | mycons : forall (y : A),
                y = x -> list_of_only A x -> list_of_only A x.
```

[1] Normally, we would be using **Set** here instead of **Type**; the difference is subtle and not terribly important for an introduction. Later in the text **Prop**, a different kind of **Type**, will be used as well. A great resource on the differences between **Set**, **Prop**, and **Type** in Coq is http://adam.chlipala.net/cpdt/html/Universes.html.

Let's `Check` all of these new definitions to make sure they match our intuition.

```
Check list_of_only.
list_of_only
 : forall A : Type, A -> Type

Check mynil.
mynil
 : forall (A : Type) (x : A), list_of_only A x

Check mycons.
mycons
 : forall (A : Type) (x y : A), y = x -> list_of_only A x -> list_of_only A x
```

3.3 The First Proof

It is easy to make an empty **ListOfOnly** type—it is only necessary to supply values for `A` and `x`.

```
Definition empty_list_of_0s := mynil _ 0.
Check empty_list_of_0s.
empty_list_of_0s
 : list_of_only nat 0
```

In order to make a non-empty list of only zeros, we must provide a proof that the element we are adding is a zero. This will amount to proving that 0 = 0, so let's make a proof for that.

```
Theorem zero_equals_zero : 0 = 0.
 ============================
 0 = 0
```

To start a proof, we use the keyword `Theorem`, followed by the name of the theorem, and then (thanks to dependent types) we state what we intend to prove as the *type* of the theorem. After seeing this statement, Coq transforms into proof mode. The current proof is displayed in a helpful form, with any hypotheses above the line of =s, and the final goal to be proved below the line. Here, we have no hypotheses to use, and we must prove 0 = 0.

To begin a proof, it is customary (but unnecessary) to input the sentence `Proof.`; however, all proofs must end with the sentence `Qed.`. Inside of proofs we have a language of *tactics* available to us to help us towards our final goal.

Zero is trivially equal to zero by the reflexive property of equality, and there is a tactic to show that:

```
reflexivity.
```

No more subgoals.

And then, to finish it off,

```
Qed.
```

zero_equals_zero is defined

This was a complete proof, though it was extremely trivial! This new proof can be used wherever 0 = 0 is required, and therefore we can define lists of only zeros with nonzero length:

```
Definition singleton_list_of_0s :=
  mycons nat 0 0 zero_equals_zero empty_list_of_0s.
Check singleton_list_of_0s.
```

singleton_list_of_0s
: list_of_only nat 0

Again, it is not necessary to explicitly provide terms when Coq can infer them:

```
Definition length_2_list_of_0s :=
  mycons _ _ 0 zero_equals_zero singleton_list_of_0s.
Check length_2_list_of_0s.
```

length_2_list_of_0s
: list_of_only nat 0

3.4 Recursive Functions

Now let's say we needed a way to convert between our new **ListOfOnly** datatype and Coq's **list** datatype; how would we go about doing that? With a recursive function, of course! Below is a function that does just that, and it is explained in the paragraphs that follow:

```
Require Import List.


Fixpoint to_list (A : Type)
```

```
                 (x : A)
                 (l : list_of_only A x)
                  : list A :=
  match l with
    | mynil => nil
    | mycons head _ tail => head :: to_list _ _ tail
  end.
```

To use Coq's list features, we first must import the `List` library.

Instead of `Definition`, recursive functions are defined with the keyword `Fixpoint`. Arguments are given in parentheses, and the return type comes after the colon. Pattern matching is done using the `match` keyword, and it has a style similar to most functional programming languages. Each of the constructors for a `list_of_only` type is broken down and used to generate a `list`, which is built using the constructors `nil` and `::`.

Now that we have such a function, let's see if it works how we expect it to:

```
Definition length_3_list_of_0s :=
  mycons _ _ 0 zero_equals_zero length_2_list_of_0s.

Eval compute in to_list _ _ length_3_list_of_0s.
```

```
= 0 :: 0 :: 0 :: nil
 : list nat
```

Indeed it does! The command `Eval` evaluates a given expression, and it requires an evaluation method (`compute` is one of them) and a term to evaluate (in our case a call to our newly-defined function).

Now suppose we wanted to make sure that our function did its job correctly. We can create a function that, when passed a list, creates a proposition that corresponds to the idea that "all elements of this list are the same":

```
Fixpoint all_are_same (A : Type) (l : list A) : Prop :=
  match l with
    | nil => True
    | x :: nil => True
    | x :: xs => x = (hd x xs) /\ (all_are_same _ xs)
  end.
```

Here we have a recursive function that takes a list and returns a logical proposition (denoted as `Prop`) that, if provable, ensures that all the elements of a list are the same. Empty and single-element lists obviously fit this criterion and return the trivially-provable proposition `True`, but lists of size ≥ 2 must have their first two elements agree, in addition to having their tails recursively satisfy this property. The `hd` function gets the head of its second argument.

```
Eval compute in all_are_same _ (to_list _ _ length_3_list_of_0s).
= 0 = 0 /\ 0 = 0 /\ True
 : Prop
```

The `/\`s are the ASCII equivalent of $\wedge$, and it is easy to see that this proposition can be proved.

3.5 The Taming of the Proof

But what if we want to prove something stronger than that? What if we wanted to ensure that *every time* we convert a **ListOfOnly** to a plain old **list** that it *always* respects the property `all_are_same`? Time for a non-trivial proof!

```
Theorem to_list_doesnt_do_anything_weird :
  forall (A : Type) (x : A) (l : list_of_only A x),
  all_are_same _ (to_list A x l).
Proof.
 ============================
 forall (A : Type) (x0 : A) (l : list_of_only A x0),
 all_are_same A (to_list A x0 l)
```

If we can manage to prove this goal, it would *certify* that our `to_list` function will always do the right thing, no matter the list that is given to it. The first step is to use the `intros` tactic, which will move all of the $\forall$-quantified variables above the line so that we can use them to prove the final `all_are_same` statement.

```
intros.
A : Type
x0 : A
l : list_of_only A x0
```

```
============================
 all_are_same A (to_list A x0 l)
```

If we were trying to prove this theorem on paper, we would need to start using mathematical induction at this point, since `list_of_only` is an inductively-defined data structure. Luckily for us, Coq supports induction as well, and each time an inductive data structure is defined Coq (for the most part) knows out how to do induction on it. `induction l.` will break down `l` into the two ways it could have been created:

```
A : Type
x0 : A
============================
 all_are_same A (to_list A x0 (mynil A x0))

subgoal 2 (ID 41) is:
 all_are_same A (to_list A x0 (mycons A x0 y e l))
```

For the first time we have more than one subgoal to prove, and we start with the case where `l` was `mynil`. The simplest way to finish off this subgoal is to run the `all_are_same` function. This can be done with the `simpl.` tactic, which is a method of evaluating expressions (just like `compute`). After running this, the goal becomes `True`, which is always true. An appeal to the tactic `trivial.` finishes off this subgoal and leaves us with the next:

```
A : Type
x0 : A
y : A
e : y = x0
l : list_of_only A x0
IHl : all_are_same A (to_list A x0 l)
============================
 all_are_same A (to_list A x0 (mycons A x0 y e l))
```

Now we have an inductive hypothesis to work with (`IHl`), among other things. Again using the `simpl.` tactic to evaluate the goal as much as possible, we see that we're stuck; the `all_are_same` function could not execute completely becuase it doesn't know exactly what the shape of `l` is, so we should break down `l` into the two possibilities: `mynil` and `mycons`. This is done with the `destruct l.` tactic, which will turn our single subgoal into two subgoals:

simpl. should finish off the mynil case again, converting the goal to True. trivial. completes this subgoal.

Now we are left with the final subgoal. We know that the list is built up using mycons and we have a helpful induction hypothesis when we need it. Using simpl. to unwind the match statement in the goal, we are left with the conjunction of an equality and another match statement.

Looking in our hypotheses above the line, we see that there is a way to rewrite the seemingly-unprovable y = y0 with x0 = x0. The subst. tactic does this for us automatically. Now, we have a provable equality and a match statement as our goal. Curiously, a tactic called auto. (which tries a few different tactics in an effort to prove the goal) will complete successfully here and solve the entire theorem for us. "But why?", you might ask. Enter Undo. to back up if you tried the auto tactic, and let's discover why our theorem is easily provable from this point.

To prove a conjunction, we must prove both parts of it separately. The split. tactic will allow us to do just that. The first subgoal is proved—you guessed it—using the reflexivity. tactic.

Proving the match case looks to be much harder. But why was the auto. tactic able to figure it out for us? It was because we already had it proven for us—our hypothesis IHl in fact is a complete proof of our goal, which makes this subgoal trivially true! But this is very hard to see at a glance, so let's evaluate IHl to see why. Enter the command unfold all_are_same in IHl. to "unfold" the definition of the all_are_same function. Now enter simpl in IHl. to partially evaluate the IHl hypothesis. Now we can see why IHl proves our goal for us: it says exactly what to do (sadly, in a not-so-elegant way) to prove the exact same match statement that we have in our goal. When we have a hypothesis that proves the goal, we can use the tactic assumption. to complete it. Qed. finishes our proof.

3.6 Quoi ?

As this paper is meant to show the power of dependent types in Coq and not to be an in-depth tutorial of Coq, this larger proof can certainly be confusing. Curious readers are encouraged to consult the Coq Documentation [4] for specifications of what individual tactics do and the Coq Tutorial [8] for a gentler introduction to proving theorems using those tactics.

If the reader is willing to trust that the logic of Coq is sound and that all proofs constructed in it are valid, we have just proved without a doubt that our definition of the **ListOfOnly** type coupled with our `to_list` function work perfectly together. It took some effort, but it gave us absolute trust in our software, which in some situations is completely worth it.

4 Get Thee to a Summary

Dependent types are perfect for proofs. When expressions are allowed in types, the type of a proof term has enough power to describe exactly what the proof proves. Proving a proposition P involves nothing more than creating an expression of type P. Coq provides tactics that help to incrementally build up a term of the intended type; there is no magic. Running `Print to_list_doesnt_do_anything_weird.` will show that—the `forall`-quantified variables in the type of the proof were replaced with a function (the `fun`) taking those variables as arguments—and you can see its type at the very end after the colon. With dependent types, the Curry-Howard correspondence really shines, and it gives rise to powerful theorem proving capabilities.

Appendix A: Vector

```
Inductive Vector (A : Type) : nat -> Type :=
  | vnil  : Vector A O
  | vcons : forall (n : nat), A -> Vector A n -> Vector A (S n).

Definition one_two_three := vcons nat 2 1 (vcons nat 1 2 (vcons nat 0 3 (vnil nat))).
Check one_two_three.

Definition one_two_three2 := vcons _ _ 1 (vcons _ _ 2 (vcons _ _ 3 (vnil _))).
Check one_two_three2.
```

Appendix B: ListOfOnly

```
Inductive list_of_only (A : Type) (x : A) : Type :=
  | mynil  : list_of_only A x
  | mycons : forall (y : A), y = x -> list_of_only A x -> list_of_only A x.

Check list_of_only.
Check mynil.
Check mycons.

Definition empty_list_of_0s := mynil _ 0.
Check empty_list_of_0s.

Theorem zero_equals_zero : 0 = 0.
Proof.
  reflexivity.
Qed.

Definition singleton_list_of_0s := mycons nat 0 0 zero_equals_zero empty_list_of_0s.
Check singleton_list_of_0s.

Definition length_2_list_of_0s := mycons _ _ 0 zero_equals_zero singleton_list_of_0s.
Check length_2_list_of_0s.

Require Import List.

Fixpoint to_list (A : Type) (x : A) (l : list_of_only A x) : list A :=
  match l with
    | mynil => nil
    | mycons head _ tail => head :: to_list _ _ tail
  end.

Definition length_3_list_of_0s := mycons _ _ 0 zero_equals_zero length_2_list_of_0s.
Eval compute in to_list _ _ length_3_list_of_0s.

Fixpoint all_are_same (A : Type) (l : list A) : Prop :=
  match l with
    | nil => True
    | x :: nil => True
    | x :: xs => x = (hd x xs) /\ (all_are_same _ xs)
  end.


Eval compute in all_are_same _ (to_list _ _ length_3_list_of_0s).

Theorem to_list_doesnt_do_anything_weird :
  forall (A : Type) (x : A) (l : list_of_only A x),
    all_are_same _ (to_list A x l).
Proof.
  intros. induction l.
    simpl. trivial.
    simpl. destruct l.
      simpl. trivial.
      simpl. subst. split.
        reflexivity.
        unfold all_are_same in IHl. simpl in IHl. assumption.
Qed.

Print to_list_doesnt_do_anything_weird.
```

References

[1] "Agda". http://wiki.portal.chalmers.se/agda/pmwiki.php.

[2] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development*. Springer Science & Business Media, 2004.

[3] "Coq". https://coq.inria.fr/.

[4] "Coq Documentation". https://coq.inria.fr/documentation.

[5] Thierry Coquand and Gérard Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2-3 (Feb. 1988), pp. 95–120.

[6] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Vol. 417. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1990, pp. 50–66.

[7] Robert Harper. "Lecture 07 – Dependent Types". http://scs.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=ba77d4f6-3fbe-42cd-90c8-8265af95e508.

[8] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. "The Coq Proof Assistant A Tutorial". http://flint.cs.yale.edu/cs430/coq/pdf/Tutorial.pdf.

[9] "Idris". http://idris-lang.org/.

[10] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.