dIFP - 2013/2014

Anders Konring, 20105366

January 16, 2014

Achilles and the Tortoise meet Genie of the lamp. Achilles is granted 3 wishes...
Achilles: "Wow! But what should I wish? Oh, I know! It's what I thought of the first time I read the Arabian Nights - I wish I had a hundred wishes instead of just three."
Genie: "I am sorry, Achilles, but I don't grant meta-wishes."
Achilles: "I wish you'd tell me what a 'meta-wish' is!"
Genie: "But THAT is a meta-meta-wish, Achilles - and I don't grant them, either..."[1]

1 Introduction

The *Fibonacci sequence* is a term introduced by *Édouard Lucas* in the nineteenth century and it describes a sequence of numbers. *Leonardo of Pisa* discovered the sequence when trying to estimate an ideal rabbit population and wrote about his findings in his book *Liber Abaci* in 1202. But the sequence appears in Indian mathematics much earlier and can be traced back to around 200 BC.[2]

This project has a twofold purpose. The main objective is to exercise and analyze different proof techniques in *Coq*. Furthermore the goal is to gain familiarity with the *Fibonacci* sequence by proving several identities related to the sequence using the power of *Coq Proof Assistant*.

1.1 Files

1.1.1 fibonacci.v

This file contains a combination of proofs and functions which we have encountered during the course. These tools come in handy when proving different lemmas. The first main proposition is the equivalence of the two specifications *specification_of_fibonacci* and *specification_of_fibonacci_alt*. D'Ocagne's identity is proven using *Finnr the Finder's* identity and the file concludes with a proof of Cassini's identity both for even and odd numbers.

1.1.2 matrix.v

The matrix file operates with 2x2-matrices. The basic operations; addition, multiplication, exponentiation and transposition are introduced in this file. Inspired by the dProgSprog notes the file contains implementations of some nontrivial properties of matrices and their operations. The end of the file treats the relation between matrices and the *Fibonacci* sequence. It is quite surprising that *Cassini's* identity is simple to prove using matrix exponentiation of a special matrix.

[1] Godel, Escher, Bach
[2] Wikipedia

Genie hates to see Achilles so disappointed and pulls some strings to grant Achilles' wish...
Genie: "This is my Meta-Lamp..."
(Genie rubs the Meta-Lamp, and a huge puff of smoke appears. In the billows of smoke, they can all make out a ghostly form towering above them.)
Meta-Genie: "I am Meta-Genie. You summoned me, O Genie? What is your wish?"
Genie: "I have a special wish to make of you, O Djinn and of GOD. I wish for permission for temporary suspension of all type-restrictions on wishes, for the duration of one typeless Wish. Could you please grant this wish for me?"
Meta-Genie: "I'll have to send it through Channels, of course. One half a moment, please..."
After a short period of time Meta-Meta-Genie appears...

2 Functions and Theorems

2.1 Specifications of Fibonacci

The *Fibonacci* sequence is generated by a recursive function or in Coq's case a *Fixpoint*. The *specification_of_fibonacci* is a logical proposition that takes an implementation of a *Fibonacci* function as argument i.e. it is defined with type *(nat → nat → prop)*. In the same manner *specification_of_fibonacci_alt* is an alternative specification inspired by *Finnr the Finder's specification_of_fibonacci* from the dPropSprog exam 2013. During the course we have been introduced to several implementations of the *Fibonacci* function and proved that they indeed satisfy the specification *specification_of_fibonacci*.

2.2 Using nat_ind

nat_ind is defined in the Prelude and thus is the default induction principle used in Coq. This induction principle is a copy of *Peano's* ninth axiom. Although this principle i really powerful, another induction principle appears to be very handy in proving properties related to the *Fibonacci* sequence. The reason for this is found in the way *Fibonacci* functions are defined. The recursive call(s) uses the last two calculated *Fibonacci* numbers to synthesize a new *Fibonacci* number. The difference between *nat_ind* and *nat_ind2* is revealed simply by checking their type.

- nat_ind
  $\forall P : nat \to Prop,$
  $P(0) \to (\forall n : nat, P(n) \to P(Sn)) \to \forall n : nat, P(n)$

- nat_ind2
  $\forall P : nat \to Prop,$
  $P(0) \to P(1) \to (\forall i : nat, P(i) \to P(Si) \to P(S(Si))) \to \forall n : nat, P(n)$

This shows that we have two hypotheses to apply in proofs using *nat_ind2* instead of only one. One usually refers to the term *complete induction* to indicate that the statement holds $\forall k \leq n$ but this case only requires the statement to hold for the two predecessors of n. Of course we have to prove two base cases as well to properly bootstrap this induction proof.

2.3 Equivalent Specifications

A function of type *nat* → *nat* satisfies the default specification if and only if the same function satisfies the alternative specification. The bidirectional implication is represented in Coq by a conjunction of two logic implications.
In *specification_of_fibonacci* → *specification_of_fibonacci_alt* requires our *nat_ind2* to be able to apply the *H_fib_ic* in the induction case. The proof uses the two induction hypotheses introduced by the modified induction principle and starts out by proving each base case respectively. To show *specification_of_fibonacci_alt* → *specification_of_fibonacci* one may realize that the default specification is a simple edition of the alternative one. Choosing $p = n$ and $q = 1$ the alternative specification yields $F_{n+1} * F_2 + F_n * F_1 = F_{n+1} + F_n = F_{n+2}$ which is exactly default specification.

2.4 D'Ocagne's Identity

When proving *d'Ocagne's* identity one may observe that this property is really just a special case of *Finnr the Finders* proposition better known as *about_fib_of_a_sum*.

$$\forall p, q : F_{p+q+1} = F_{p+1} + F_{q+1} + F_p + F_q$$

Letting $p = n + 1$ and $q = n$ we obtain

$$\forall n : F_{n+1+n+1} = F_{2(n+1)} = F_{n+2} + F_{n+1} + F_{n+1} + F_n = F_{n+1}(F_{n+2} + F_n)$$

In other words when proving this identity it is only necessary to apply *about_fib_of_a_sum* with arguments $(S\ n)$ and n to obtain the identity and thus the proof is accomplished.

2.5 Cassini's Identity

Cassini's identity is often stated[3] like this

$$F_{n-1}F_{n+1} - F_n^2 = (-1)^n$$

The file has split the proof into two parts. One proof for odd n and one proof for even n respectively. If $n = 2k + 1$ it yields the identity described in the file for odd numbers.

$$F_{2k}F_{2(k+1)} + 1 = F_{2k+1}^2$$

and setting $n = 2(k + 1)$ yields the identity for even numbers

$$F_{2k+1}F_{2(k+1)+1} = F_{2(k+1)}^2 + 1$$

When starting out proving the identity it is tempting to use *nat_ind2* since this induction principle does work very well in this context. However, in this proof it is not necessary to complicate things. The author found himself struggling with the proof and working with bloated expressions until he took another and simpler approach using *nat_ind* and the *binomial theorem*.

The first application of *fib_ic* in the inductive case makes sure that what is on the left hand side is actually recognized as part of the *binomial theorem*. From there it is all about applying the induction hypotheses on the right hand side. Afterwards the *binomial theorem* is used to make an assertion *H_bin*. This assertion turns out to be provable and the proof is complete when *binomial2* is applied in the end.

[3] Wolfram MathWorld

3 Matrices and Fibonacci

The file *matrix.v* introduces addition, multiplication, exponentiation and transposition of matrices. A matrix is a simple inductive type with one constructor
$| \; Matrix22 \Rightarrow nat \rightarrow nat \rightarrow nat \rightarrow nat \rightarrow matrix22$.

3.1 Matrix Exponentiation

In the dProgSprog notes several interesting identities are stated and proved. This project offers some of those proofs implemented in Coq. Proposition *exponentiation_of_a_matrix* (or proposition 14) states

$$\forall n : \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^n = \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix}$$

The proof in Coq is short and uses the standard induction.

Likewise *about_matrix_exponentiation* (or proposition 29) is proved

$$M \times M^n = M^n \times M$$

This shows the commutativity of matrix exponentiation and will be quite handy in later induction proofs.

3.2 Matrix Transposition

matrix_transposition_and_exponentiation_commutes (or Proposition 38) shows that transposition and exponentiation commutes.

$$\forall M \forall n : (M^n)^T = (M^T)^n$$

3.3 Matrices and Fibonacci

The relationship between Fibonacci sequence and 2x2-matrix exponentiation can be stated like this

$$\forall n : \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

This identity is proved in *matrix_exponentiation_and_fibonacci* and this is quite a beautiful identity. Furthermore this theorem is developed into a new version of the *Fibonacci* function *fib_v3* by choosing the coordinate a_{01} to be the output of the function. By showing functional equality between this implementation and the direct style implementation it is easy to show that *fib_v3* satisfies the default specification of *Fibonacci* too.

3.4 Cassini's Identity with Matrices

Taking the above one level further we can actually state Cassini's identity using matrices and the determinant.

$$F_{n-1}F_{n+1} - F_n^2 = det \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = det \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = (-1)^n$$

The determinant is implemented but has one unfortunate feature. The image of the determinant function is $\mathbb{Z}$ and not $\mathbb{N}$. This means that the the Coq implementation of the determinant function will yield zero if the determinant is indeed non-positive.

The proof of the above statement is given for even n but did not succeed for odd n. Why? Well, we know from the above statement that for every odd n the determinant will be -1. Since $-1 \notin \mathbb{N}$ Coq cannot prove this identity in this manner.

After a while Genie returns and is now able to grant Achilles meta-wishes.
Achilles: "That gives me an idea for my wish."
Tortoise: "Oh, really? What is it?"
Achilles: "I wish my wish would not be granted..."

4 Conclusion

4.1 Working with Coq

Coq Proof Assistant is very different from other programming platforms that I have ever worked with. In the beginning of the course it felt very much like a slow but strict way to formalize proofs. Now the idea of seeing a proof as a program and a proposition as a type makes much more sense. Fx *Print matrix_multiplication_is_associative* yields 500 lines of Coq code. This shows that whenever we write tactics to obtain a goal we are really just creating a program. In the same manner we can look at a proposition or statement of a theorem as a type and the proof as exactly the program for which the type-checker is satisfied.

When proving in Coq the tactics enforce a great deal of discipline and it is very easy to follow each step taken in the proofs. In ordinary math books it is sometimes unclear which steps the author took and which assumptions he made in order to get from one expression to another. In Coq every expression or goal is a result of applying some lemma or theorem already proven and the rules of inference and the type system helps the author avoid incorrect proofs.

The power of Coq really shines in proofs using induction. However, sometimes it can be difficult to know in which direction to go during a proof and which induction principle to apply. In this course we have only had a limited supply of tactics available. This has indeed helped simplify the Coq language and flattening the learning curve and I think with a little more experience and a couple of more tactics it can be a really efficient tool.