Verifying programs and proofs
Part I. describe algorithms

Yves Bertot

October 2020

1 Motivating introduction

Starting in the 1940s, it was understood that one could use computers to do math. People who don't know much about mathematics usually think that the purpose of mathematics is to compute numbers and obviously computers can help compute. But there is more in math than just computing numbers, there is also reasoning.

Reasoning follows rules, the rules of logic. If you want to convince somebody, you need to follow these rules, otherwise the other side can always claim that they don't trust you. The rules must be quite dumb, so that nobody can reject them, however stubborn they are. The rules are so dumb, that actually a computer can check that they are applied correctly.

If Alice needs to check that Paul works well, it is very difficult for Alice if what Paul has to do is long and boring, so that Alice only has dumb rules to check. But a computer can help, if it can recognize whether the rules are applied or not, because a computer never gets tired.

This is the game: reduce mathematics to something dumb, so that a computer can check it, and then rely on the computer.

Programming and software engineering can benefit from this, too. To make sure that a program does what it is meant to do, we need to check all cases. We can try to do it with massive testing, but testing can only catch mistakes somehow at random. On the other hand, we can reason about what program do and verify that they behave correctly in all possible cases.

This approach raises a lot of interest in industry, especially for companies that are very vulnerable to software or hardware bugs. For instance, Intel lost billions of dollars because the algorithm for division that they engraved in the Pentium (circa 1998) was faulty. They now hire specialists in computer-based proof (like the author of [5]). Aeronautics companies more and more replace physical objects (cables and rods) by networks and software in the command system of planes, but they have to convince certifying authorities that the software is as safe as or safer than the physical devices they replace: they are interested in hiring software engineers that know how to guarantee the correctness of their software. In France, the agency for research on nuclear energy also invests a lot of efforts on these approaches and the the national agency for software system security also publishes guidelines to use proof systems.

There are several approaches to verifying the quality of software. Some approaches concentrate on programming languages that are closer to widespread langages like C or Java. Other rely on programming languages that are easier to reason about in an efficient fashion. We will use the latter approach and the language we consider is called a *functional programming language*.

2 Starting the Coq system

To work with Coq, you can start with three commands. The first, called `coqc` is used like a compiler; the second, called `coqtop` was initially meant to work in a terminal window: you type in commands and it prints out answers; the third one, called `coqide`, is an interactive development environment: it keeps a window where all your commands are recorded and other windows where the most recent results from the tool are displayed. You are advised to use `coqide`, but I will be using an alternative approach, based on a text editor called `Emacs` and a specialized mode (called `Proof General`) that encapsulates `coqtop` and provides approximately the same features as `coqide`.

When using `coqide`, you must be aware that the commands you type are only processed if you explicitely require processing to happen, either by clicking on an arrow in the top row of the window or by typing in the appropriate combination of special characters. The tool tries to keeps a precise account of what you have done and how the state of the system has been modified, so that you are not allowed to modify a command that is recorded as "processed" (processed commands appear on different background). To modify a processed command, you first need to ask the tool to "unprocess" it. This is also possible using one of the arrows in the top row of the window.

These notes will describe only a subset of the features provided by the tool. If you want to know more, or if something is not clear, you should look at the following pointer:

`http://coq.inria.fr`

3 First steps with the programming language

The Coq system contains a programming language that is very simple but also quite unconventional.

For start, it is interesting to require that the system loads objects that have already been defined. In our first experiments, we will use boolean values, natural numbers, and lists. So, we should start with the following command:

```
Require Import Arith List Bool NPeano.
```

Natural numbers are already unconventional for people trained to use computers: they are all positive.

In a programming language, you need to be able to define new things, you need to be able to test data and to take decision based on the test results, and you need to be able to repeat operations. This is what we will see in the following examples.

To define new things, you use the keyword `Definition`.

```
Definition four := 4.

Definition polynom1 (x:nat) := x * x + 3 * x + 1.
```

Types play an important role in Coq programming: most things have a type. For instance, by default the numbers 0, 1, 2, ... have type `nat`. When you write a function, like the `polynom1` function above, you usually give the type of the expected arguments. This is similar to conventional programming languages.

At any time you can use a command to verify if some formula is well formed. This command is called `Check`.

```
Check polynom1 (polynom1 (polynom1 (polynom1 (polynom1 4)))).
```

This command performs no computation. It only checks that you apply functions to arguments of the right type.

To evaluate a value, you can use a command called `Compute`.

```
Compute polynom1 (polynom1 4).
```

With the kind of number representation used in our first steps, computation is very slow and consumes a lot of memory and you should not try to compute the big nesting of terms that was use for the `Check` command. But there are other advantages, which we will see when reasoning about our programs.

In this programing language, we can also buid functions without naming them. Such functions only exist during the time we consider them.

```
Check (fun x => x * x, fun x => x * polynom1 x).
```

We can also define things locally, so that they are forgotten after the command where they are supposed to be used. The construct is called `let`.

```
Check let x := 3 * 4 in x * x * x.
```

You can also verify that a command will fail. For instance, we know that `x` is no longer defined after the previous command and we can verify that.

```
Fail Check x.
The command has indeed failed with message:
=> Error: The reference x was not found in the current environment.
```

3.1 Simple data structures and pattern-matching

In these first steps we will only use 5 different families of datatypes.

3.1.1 Natural numbers

The first datatype will be the datatype of natural numbers called `nat`. Defining a datatype relies on giving all possible cases in this type. For natural numbers, the definition says that there are only two cases: a natural number is either the number 0 or the successor of another natural number. The internal name for the number 0 is written `O` (upper-case o). The internal name for the other case is written `S`. This `S` can actually be used like a function that takes as argument a natural number.

The notations with digits, like 13, 5, 8, are only notations. They are provided by the Coq system to display data in a more understandable way, but the data that is really manipulated internally is really of the form `S (S (S (S (S O))))` for 5. You can experiment with this as in the following commands:

```
Check (S O).
1 : nat
```

This datatype is not oriented towards computer efficiency, but towards mathematical ease of reasoning. In modern mathematics, its takes three digits to write the number 100, and 4 digits to write the number 1000. But with the `nat` datatype, its takes approximately 2 memory chunks to represent the number 1 and 11 memory chunks to represent the number 10. This is inefficient, and this is the reason why it is not advise to compute large numbers with this datatype.

The datatype of natural numbers comes with a few pre-defined functions and fairly usual notations. For instance, we can compute the addition of two numbers:

```
Compute 3 + 7.
 = 10 : nat
```

There are also two functions for comparing natural numbers, with notation `<=?` and `=?`, the first computes whether the first number is less than or equal to the second number, the second computes whether the two numbers are equal. The result is a boolean value of type `bool` that we describe more in details later.

```
Compute 7 <=? 3.
 = false : bool
```

Note that the notation for comparing two numbers in a program is particular, with the question mark attached to the end. We shall later use a different notation for a different kind of comparison between numbers, which cannot be used inside programs.

3.1.2 Lists

The second family of datatypes that we will consider is the family of lists. It is very similar to the arrays that we are accustomed to use in conventional programming language. The common feature is that lists also have to contain elements that are all of the same type. The difference is that you don't have to choose in advance the size of a list.

Like for natural numbers, there are only two cases in lists: a list is either an empty list, with the internal name `nil` or it is compound object with two sub-components, with the internal name `cons`. The first component is the first element of the list, the second component is another list that contains all the other elements of the list.

The internal name `cons` is almost never used. Instead we use a notation with "`::`". So we can build a list with three elements as in the following command, but the answer of the computer uses the notation.

```
Check cons 3 (cons 2 (cons 1 nil)).
3 :: 2 :: 1 :: nil
     : list nat
```

Note that the type of the thing considered above is `list nat`. So when we consider a list, the system always tries to know what the type of elements is. We can also construct more complex data, for instance lists of lists. The following shows how to construct a list with two elements, which are both lists of natural numbers. This could be used to represent a bi-dimensional array of 2×2 elements.

```
Check cons (cons 2 (cons 1 nil)) (cons (cons 3 (cons 2 nil)) nil).
(2 :: 1 :: nil) :: (3 :: 2 :: nil) :: nil
   : list (list nat)
```

3.1.3 pairs

Lists can be used to group several things in one new thing, but they all have to be of the same type. If you want to group several things together that are from different types, you will use another datatype, called the type of pairs. For instance, if you want to group together a number and a list of numbers, you will just write as follows:

```
Check (3, 2::1::nil).
(3, 2::1::nil) : nat * list nat
```

There is only one case in this datatype of pairs.

3.1.4 boolean values

In programming we often use a datatype that has only two elements, called `true` and `false`. This datatype is called `bool`. The following command checks that `true` really exists.

```
Check true.
```

3.1.5 optional values

Sometimes, we will need to write a program that either returns exactly one value or returns an exceptional condition. For instance, one may want to have a function that takes as input a natural number and returns its predecessor if it exists. So this function would map 2 to 1 and 1 to 0. But for the input 0, we want to express that there is no number that fits. So we may choose that this function returns an element of the datatype `option nat`. This datatype has two cases, where the first case is called `Some` and it has a subcomponent which is the thing we want to return. The second case is called `None` and it has no subcomponent. So, for instance, to build an element of `option nat` we can write the following expression:

```
Check Some 3.
Some 3 : option nat
```

3.2 Observing, testing, and making decisions

If you manipulate an object in one of the datatypes, you may need to know in which case you are for the datatypes, and what are the subcomponents. To do this, you use a construct called “pattern matching”.

The first simple example is when the datatype has only one case. Assume `p` is a pair, you can write the following:

```
match p with
  (a, b) => ...
end
```

inside the expression hidden in the dots, you are allowed to use `a` to refer to the first component of `p` and `b` to refer to the second component.

For lists, it is a little more complex, because your program has to be ready to handle all possible cases in the datatype. so you need to say what to do if `p` is in the `cons` case (noted with “`::`”) and what to do if `p` is in the `nil` case.

```
match p with
  a::v => ...
| nil =>  ...
end
```

So this construct *tests* whether `p` is empty or not, takes a decision accordingly and executes the piece of code that corresponds, using the variables `a` and `v` to observe the subcomponents.

If we use the pattern-matching construct on a boolean value, we do not observe subcomponents but we still take decisions. This construct behaves exactly like an *if-then-else*. This is illustrated by the notations of the Coq system.

```
Check match 3 <=? 7 with true => 1 | false => 2 end.
if 3 <=? 7 then 1 else 2
     : nat
```

We can illustrate the behavior of the pattern-matching construct using the `Compute` command. The first command computes the comparison between two natural numbers and then takes a decision depending on the result. The second command observes a list and returns its first element. Note that the case where the list would be empty also needs to be covered in the `match` construct. This makes programming safer.

```
Compute match 3 <=? 7 with true => 1 | false => 2 end.
 = 1 : nat

Compute match 3::2::1::nil with nil => 0 | a::v => a end.
 = 3 : nat

Compute match (3, true) with
  (a, b) => match b with false => 4 | true => 2 end
 end.
 = 2 : nat
```

3.3 Defining and applying functions

We can obtain functions in two ways. Either functions were defined earlier and a name was given to them (by using `Definition` or `let`) or we build a function to use it directly. To apply a function to an argument we simply write this function on the left of the argument. All four of the following commands are correct.

```
Check polynom1 1.
polynom1 1 : nat

Check (fun x => x * x) 3.
(fun x => x * x) 3 : nat

Check (fun x => (fun y => x + y)) 3.
(fun x => (fun y => x + y)) 3 : nat -> nat

Check (fun x => (fun y => x + y)) 3 4.
```

In the last but one command, we see that the expression is a function, so it can be applied again to another number by writing it on the left of that number. This is what we do in the last command. Note that there is no need to use more parentheses.

3.4 Repeating computation

For repeating computation, the programming language of Coq is even more unconventional. The important aspect is that Coq refuses to let you define functions that may loop forever. Repetition is only allowed using recursion, and recursion is only allowed when the function's main argument gets smaller. In our first steps with the Coq system, we will only consider recursive functions that take numbers and lists as arguments.

Here is a simple example of a function with recursion. This function takes as input a list of natural numbers and adds all the elements in the list.

```
Fixpoint slnat (l : list nat) :=
 match l with nil => 0 | a::tl => a + slnat tl end.

Compute slnat (1 :: 3 :: 4 :: nil).
 = 8 : nat
```

First, note that the keyword for this definition is `Fixpoint`. You have to use this keyword if you want to define a function that has a repetitive behavior. Second, note that the function relies on a pattern matching construct to decide on what to do. In the case where the input list is empty, the computation terminates and return 0. In the case where the input list contains a first number `a` and a smaller list `tl`, the computation of a recursive call of `slnat` is required on this smaller list. This will give a result to which `a` will be added. So, we explicitely have that the recursive call is done on a smaller list.

Recursive calls do not need to be done on direct subcomponents of the initial input. For instance, the following function is accepted:

```
Fixpoint teo (l : list nat) :=
  match l with
    nil => nil
  | a::nil => a::nil
  | a::b::tl => a::teo tl
  end.

Compute teo (1 :: 3 :: 4 :: nil).
 = 1 :: 4 :: nil : list nat
```

The function `teo` takes every other element of the input lists. In the case where the input list has the shape `a::b::tl`, the recursive call is performed on the list `tl` which is visibly a subcomponent.

For recursive function that have an natural number as argument, the same structure appears. The recursive function must rely on pattern-matching to test whether the argument is 0 or not, and a recursive call is possible only when the argument is in the `S` case. When the recursive call is possible, it must be on the subcomponent (or one of its subcomponents). Here is an example, where the recursive function builds the list of natural numbers smaller than the input.

```
Fixpoint lsn (n : nat) :=
 match n with
   0 => nil
 | S p => p :: lsn p
 end.
```

```
Compute lsn 4.
 = 3 :: 2 :: 1 :: 0 :: nil
  : list nat
```

We see in this example that pattern-matching provides a handy way to compute the predecessor of a number.

We can also define functions with several arguments, but the verification about recursive calls on smaller values must rely only on one of the argument positions. The following example describes an addition function.

```
Fixpoint add (n m : nat) :=
  match m with 0 => n | S p => add (S n) p end.
```

In this function, the second argument decreases at each recursive call. If we ask for the computation of `add 3 2`, then the first recursive call requires that we compute `add (S 3) 1` and this is written `add 4 1`. The second recursive call requires that we compute `add 5 0`. In this last computation, the first branch of the pattern-matching construct is performed and the result is `5`.

4 Transforming a complex algorithm into a functional program

One of the difficulties of programming in functional programming languages is to find ways to transform repetitive processes into recursive functions. We will try to illustrate this using an algorithm that most of us know, because we learned it in elementary school: the division algorithm. let's replay this algorithm on two examples: dividing 3845 by 17 and dividing 1645 by 17.

$$\begin{array}{r|l} 3845 & 17 \\ \cline{2-2} & \end{array} \qquad \begin{array}{r|l} 1645 & 17 \\ \cline{2-2} & \end{array}$$

Usually, we start by pushing the number 17 to the end of the dividends 3845 and 1645: because 17 only has two digits, we take the last two digits of the dividends, 38 for 3845 and 16 for 1645 and then we find the largest number a such ($a \times 17$ is smaller than or equal to 38 or 16. In the first case with $a_1 = 2$ because $2 \times 17 = 34 <= 38 < 3 \times 17 = 51$, in the second case we find $a_0 = 0$. Then we compute the remainder by removing $a \times 17$ the considered number. In the first case we have $38 - 2 \times 17 = 4$, in the second case we keep 16. We write a_1 or a_2 on the right part of our division figure as the beginning of our quotient: so the quotient start with 2 in the first case and with 0 in the second case (often, we don't bother writing this 0).

$$\begin{array}{l|l} \texttt{3845} & \texttt{17} \\ \cline{2-2} \texttt{ 4} & \texttt{2} \end{array} \qquad \begin{array}{l|l} \texttt{1645} & \texttt{17} \\ \cline{2-2} \texttt{16} & \end{array}$$

The next step is to consider the number obtained by taking the first digit that was not yet considered in the dividend and stick it to the right of the remainder. In our examples, the first digit is 4 and we stick it to the right of 4, which was the remainder in the first case, to obtain 44, and to the left of 16, which was the remainder in the second case, to obtain 164.

$$\begin{array}{r|l} \texttt{3845} & \texttt{17} \\ \cline{2-2} \texttt{44} & 2 \end{array} \qquad \begin{array}{r|l} \texttt{1645} & \texttt{17} \\ \cline{2-2} \texttt{164} & \end{array}$$

Now, we start repeating the same operations: we want to find the largest b such that $b \times 17$ is smaller than or equal to 44 in the first case and smaller than or equal to 164 in the second case. We obtain $b_1 = 2$ in the first case and $b_2 = 9$ in the second case, and the new remainders are 10 and $164 - 9 \times 17 = 164 - 153 = 11$ in the second case. The quotient becomes 22 in the first case and 9 in the second case.

$$\begin{array}{r|l} \texttt{3845} & \texttt{17} \\ \cline{2-2} \texttt{44} & 22 \\ \texttt{10} & \end{array} \qquad \begin{array}{r|l} \texttt{1645} & \texttt{17} \\ \cline{2-2} \texttt{164} & 9 \\ \texttt{11} & \end{array}$$

Then we repeat the same steps again and by taking again the first digit that was not yet considered in the dividend and add it to the left of the remainder. So, we take 5 and stick it to 10 to obtain 105 in the first case, and to 11 to obtain 115 in the second case.

$$\begin{array}{r|l} \texttt{3845} & \texttt{17} \\ \cline{2-2} \texttt{44} & 22 \\ \texttt{105} & \end{array} \qquad \begin{array}{r|l} \texttt{1645} & \texttt{17} \\ \cline{2-2} \texttt{164} & 9 \\ \texttt{115} & \end{array}$$

Division by 17 yields a digit that is 6 in both cases, so the new quotient is 226 in the first case, with a remainder of 3, and 96 in the second case, with a remainder of 13.

$$\begin{array}{r|l} \texttt{3845} & \texttt{17} \\ \cline{2-2} \texttt{44} & 226 \\ \texttt{105} & \\ \texttt{3} & \end{array} \qquad \begin{array}{r|l} \texttt{1645} & \texttt{17} \\ \cline{2-2} \texttt{164} & 96 \\ \texttt{115} & \\ \texttt{13} & \end{array}$$

We can check that the result is correct by computing the following equations:

$$226 \times 17 + 3 = 3845 \qquad 96 \times 17 + 3 = 1645.$$

Now, let's work on representing this algorithm in the Coq system. This algorithm works with numbers as if they were sequences of digits, so we will use lists of digits to represent the numbers. In this algorithm, we add digits to the right (for instance, we stick 4 to the right of 16 to obtain 164) but in a list, we add elements to the left, so we will use the convention of representing a number by a list of digits where the digits come in reverse order. To represent the digits, we will simply use natural numbers. So for instance, the number 239 will be represented by the list (9::3::2::nil).

To understand how to represent the algorithm recursively, we need to look at one of the last repetitions of the algorithm. What is happening there, for instance when dividing 3845 by 17, we look specifically at 5, and we use the remainder of dividing 384 by 17, which is 10, and the quotient of that division, which is 22. So we can say: We take the list `(a::l)` (where `a` is 5 in the example and `l` is `(4::8::3::nil)`) and we divide `l` by the divisor (let's call it `d`), thus obtaining a quotient q and a remainder r. So in our presentation of the algorithm we will write something like that:

```
match input with
  a::l => let (q, r) := divide l d in
```

The next step is to stick the number 5 to 10, so with our variables, we stick `a` on the left of `r`, which brings us to consider the list `a::r` and to perform a "small division" on this list: we know that this small division will return only one digit, that is a number smaller than 9, which will be represented as a natural number. So, for now, we assume that there exists another function to perform the small division, which returns just the required number. Let's call this small division function `small_divide`, we want to compute the following expression:

```
let n := small_divide (a::r) d in
```

Thus we give the name `n` to the result of the small division. Now we want to use this name `n` in two ways. First we want to stick `n` to the left of the previous quotient `q` to obtain the new quotient, second we want to remove `n * d` from `(a::r)` to obtain the new remainder (in the example, this corresponds to computing $105 - 6 \times 17 = 3$. For this task, we assume we have two function subtract and multiply that act on lists of digits to represent the arithmetic operations. So we complete the work by writing the value that is returned:

```
  (n::q, subtract (a::r) (multiply (n::nil) d))
```

In all, the recursive case of the algorithm is written as:

```
match input with
  a::l => let (q, r) := divide l d in
  let n := small_divide (a::r) d in
  (n::q, subtract (a::r) (multiply (n::nil) d))
```

Now, we have to consider the other case for the dividend. This other case is when the number is represented by the empty list. What number is represented by the empty list? Actually, it is 0. So if the list is empty, we want to divide 0 by a given divisor `d`, the obvious result is the quotient 0 and the remainder 0, so we can simply return the following value, where `nil` is used to represent 0 in the remainder part of the result.

```
  (0, nil)
```

In the end, the whole division algorithm is described as follows:

```
Fixpoint divide (input d : list nat) : list nat * list nat :=
  match input with
    a :: l => let (q, r) := divide l d in
    let n := small_divide (a::r) d in
    (n::q, subtract (a::r) (multiply (n::nil) d))
  | nil => (nil, nil)
  end.
```

Note that this recursive function has a recusive call on `l` as first argument, which is a sub-component of `input` as decribed by the pattern-matching construct.

It remains to describe the functions that are being used in the algorithm. Let's look at `small_divide`. We know we want to find a number that is smaller than 10, so we can perform a little exhaustive search to find the correct one, starting at 10. This can be written as follows:

```
Fixpoint small_div (k : nat) l q :=
  match k with
  | S k' => if l_le (n_l_mul k q) l then k else small_div k' l q
  | _ => 0
  end.

Definition small_divide l q := small_divide_aux 9 l q.
```

Again, we have to define a new auxiliary function, `less_than`, which checks whether the number represented by a list of digits is strictly smaller than another. Here is an example implementation:

```
Fixpoint is_zero (l : list nat) :=
  match l with a :: tl => (a =? 0) && is_zero tl | nil => true end.

Fixpoint eq_number (l1 l2 : list nat) :=
  match l1, l2 with
  | a::t1, b::t2 => (a =? b) && eq_number t1 t2
  | _, _ => is_zero l1 && is_zero l2
  end.

Fixpoint less_than (l1 l2 : list nat) :=
  match l1, l2 with
  | a :: tl1, b :: tl2 =>
    if eq_number tl1 tl2 then a <=? b else less_than tl1 tl2
  | _, b :: tl2 => true
  | _, _ => is_zero l1
  end.
```

5 Exercises

1. Define a function that takes as input a list of numbers and returns the product of these numbers,

2. Define a function that takes a list of numbers and returns `true` if and only if this list contains the number `0`.

3. Define a function that takes as input two numbers and returns `true` if and only if these two numbers are the same natural number (such a function already exists in the Coq libraries (function `beq_nat`, but how would you define such a function), using only pattern-matching and recursion.

4. Define a function that takes a number `n` and a number `a` as input and returns a list of numbers containing `n` elements that are all `a`.

5. Define a function that takes a number `n` and a number `a` and returns the list of `n` elements a ::$a + 1$:: $\cdots$ ::$a + (n - 1)$:: `nil`.

6. Define a function that takes as input a natural number and returns an element of `option nat` containing the predecessor if it exists or `None` if the input is 0.

7. Define a function that takes as input a list of numbers and returns the length of this list. — Adde (e1 e2 : exp)

8. Can you write a function `values` that takes as input a function f of type `nat -> nat`, an initial value a of type `nat` and a count n of type nat and produces the list

   a :: f a :: f (f a) :: ...

9. To every natural number, we can associate the list of its digits from right to left. For instance, 239 should be associated to the list `9::3::2::nil`. We also consider that 0 can be mapped to `nil`. If `l` is such a list, we can consider the successor of a list of digits. For instance, the successor of `9::3::2::nil` is `0::4::2`. Define the algorithm on lists of natural numbers that computes the successor of that list.

10. Assuming that `lsuc` is the function defined at the previous exercise, define the function `nat_digits` that maps every natural number to the corresponding list of digits (naive solutions are welcome, as long as they run).

11. In the same context as the previous two exercises, define a function `value` that maps every list of digits to the natural number it represents. Thus, `value (9::3::2::nil)` should compute to 239.

12. In the same context as the previous exercises, define a function `licit` that tells whether a list of integers corresponds to the digits of natural number: no digit should be larger than 9. For instance, `licit (9::3::2::0::nil)` should compute to `true` and `licit (239::nil)` should compute to false.

13. In the same context as the previous exercises, define functions to add two lists of digits so that the result represents the sum of the numbers

represented by the lists of digits. In other words `addl (7::2::nil) (5::3::nil)` should return `2::6::nil` (Hint: you should implement the algorithm you learned in elementary school for adding numbers).

14. In the same context as the previous exercises, define a function for multiplying a list by a small natural number (by successive additions) and then a function `mull` for multiplying two lists of digits.

6 An example of a complex program

The following program computes 10 digits of π. It uses data-types and functions that have not been given above, but it could be redone as an exercise using the lists of digits from exercises 9 to 14, and adding algorithms for addition, multiplication, and division of numbers (as in section 4), as we learn them in school. Note that the function (`atanV` n x) is used to compute the following mathematical formula

$$\sum_{i=0}^{n} \frac{(-1)^n}{(2n+1)\times x^{2n+1}} = \frac{1}{x} - \frac{1}{3}\frac{1}{x^3} + \frac{1}{5}\frac{1}{x^5} + \cdots$$

From mathematics, we know that this provides good approximations of $\mathrm{atan}(\frac{1}{x})$, then π is approximated by a formula known as Machin's formula:

$$\pi = 4 \times \left(4 \times \mathrm{atan}(\frac{1}{5}) - \mathrm{atan}(\frac{1}{239})\right)$$

We obtain a rational number that is good approximation of π, to get the first 10 digits, we multiply the numerator by 10^{10} and divide by the denominator and take the integer part.

```
Require Import QArith.
Open Scope Q_scope.

Fixpoint atanx n (acc : Q) s p (y x : Z) :=
match n with O => acc
| S n' => atanx n' (acc + (s#p) * /(y#1)) (-s)%Z
   (p + 2)%positive (y * x ^ 2) x
end.

Definition atanV n x := atanx n (/(x#1)) (-1) 3 (x ^ 3) x.

Definition PI_approx :=
  Qred ((4#1) * ((4#1) * atanV 7 5 - atanV 1 239)).

Time Compute
 match PI_approx with (a#b) => Z.div (10 ^ 11 * a) (Zpos b) end.
```

7 More information

You can use the book [1] (available in French on internet, otherwise you should find English versions at the library) and the reference manual [4]. There is also a tutorial in French [7]. There are also tutorials on the web [3, 6].

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[2] B. Pierce et al. *Software Foundations* `http://www.cis.upenn.edu/~bcpierce/sf/`

[3] Y. Bertot *Coq in a Hurry* Archive ouverte "cours en ligne", 2008. `http://cel.archives-ouvertes.fr/inria-00001173`

[4] The Coq development team. *The Coq proof Assistant Reference Manual*, Ecole Polytechnique, INRIA, Université de Paris-Sud, 2004. `http://coq.inria.fr/doc/main.html`

[5] John Harrison, *Handbook of Practical Logic and Automated Reasoning'* Cambridge University Press, 2009 `http://www.cl.cam.ac.uk/~jrh13/atp/`

[6] E. Giménez, P. Castéran, *A Tutorial on Recursive Types in Coq*, INRIA, Université de Bordeaux, 2006. `http://www.labri.fr/Perso/~casteran/RecTutorial.pdf.gz`

[7] A. Miquel, *Petit guide de survie en Coq*, Université de Paris VII. `http://www.pps.jussieu.fr/~miquel/enseignement/mpri/guide.html`