Certification of Sorting Algorithms in the Coq System

Jean-Christophe Filliâtre[1] and Nicolas Magaud[2]

[1] LRI, Université Paris Sud, bât. 490,
91405 Orsay Cedex, France,
filliatr@lri.fr,
web page: www.lri.fr/~filliatr

[2] École Normale Supérieure de Lyon,
69364 Lyon Cedex 07, France
nmagaud@ens-lyon.fr,
web page: www.ens-lyon.fr/~nmagaud

Abstract. We present the formal proofs of total correctness of three sorting algorithms in the Coq proof assistant, namely *insertion sort*, *quicksort* and *heapsort*. The implementations are imperative programs working in-place on a given array. Those developments demonstrate the usefulness of inductive types and higher-order logic in the process of software certification. They also show that the proof of rather complex algorithms may be done in a small amount of time — only a few days for each development — and without great difficulty.

1 Introduction

Once a formal specification is given, we can write a program that meets the specification with mathematical rigor, each step being fully justified, leading to a correct implementation. However, the large number of cases and the tedious technical proofs — as for instance arithmetical properties — discourage the programmer most of the time. Moreover, he or she may misinterpret some semantically subtle point, as for instance the lazy evaluation of some logical connective. For those reasons, the help of a formal method to produce the proof obligations and of a proof assistant to establish them is quickly unavoidable.

We present here the proofs of correctness and termination of three sorting algorithms in the Coq system, namely *insertion sort*, *quicksort* and *heapsort*. Why sorting algorithms ? Mainly because they are short but complex programs, which cover a huge panel of program constructions — loops, recursive functions, local variables, function calls, etc. As noticed by R. Sedgewick in his book *Algorithms* [6], it is not so easy to write a correct implementation of *quicksort*, even without any optimization, and a small mistake in such a complex algorithm has immediately some catastrophic consequences.

Those case studies have demonstrated the relevance of the use of the Coq system in the proof of imperative programs correctness. In particular, we used inductively defined predicates several times in the developments. We also used higher-order to define new induction principles, or to prove the well-foundedness of some relation to establish the termination of a loop. The formal developments described in this article are freely available on the web page of Coq users' contributions (`coq.inria.fr/contribs`). The Coq system and the tactic used to prove the correctness of imperative programs are also freely available (at `coq.inria.fr`).

This article is organized as follows. The next section introduces the proof of programs in the Coq system, giving the syntax of programs and annotations, and describing the useful datatypes. Section 3 defines the libraries used in the specification of sorting algorithms, which are common to the three developments. Then Section 4 describes the three case studies in details. We conclude with a discussion about the difficulties encountered during the process of specifying and proving.

2 Certifying programs with the Coq system

The Coq system [1] is a proof assistant for the Calculus of Inductive Constructions, a logical framework extending the system F with higher-order, dependent types and a primitive notion of inductive types. One can introduce new definitions and prove facts, using an interactive prover in a natural deduction way. As a typed λ-calculus, the logic of the Coq system is naturally well-suited to prove purely functional programs. It is now also possible to establish the correctness and termination of imperative programs [2].

Programs are given in a syntax mixing functional and imperative features, close to the syntax of the Caml programming language:

$$\begin{array}{rcl} e & ::= & c \mid x \mid \mathsf{fun}\ (x:\tau) \rightarrow e \mid (e\ e) \mid \mathsf{rec}\ f:\tau = e \\ & \mid & !x \mid x := e \mid t[e] \mid t[e] := e \mid e\ ;\ e \mid \mathsf{while}\ e\ \mathsf{do}\ e\ \mathsf{done} \\ & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{ref}\ e\ \mathsf{in}\ e \end{array}$$

The base objects are the constants of the logic, written c. It means that the programs can manipulate any datatype defined within the proof assistant, and there are no particular base types — but, of course, one can use the already predefined types of the Coq system, like the type *nat* of natural number or the type Z of integers. The mutable variables and the arrays must contain only purely functional values. Therefore, the type system of the programming language is the following:

$$\tau \quad ::= \quad T \mid T\ \mathsf{ref} \mid T\ \mathsf{array} \mid \tau \rightarrow \tau$$

where T stands for any type defined in the Calculus of Inductive Constructions.

Programs are specified using pre- and post-conditions, with the classical notation of Floyd-Hoare logic $\{P\}\ e\ \{Q\}$, which can be applied to any sub-expression of the program. Those annotations may refer to the values of program variables. If x is such a variable, then x in P and Q stands for the *current* value of x. In the post-condition Q, $x_@$ stands for the value of x *before* the evaluation of e. It is also possible to refer to the value of x at a given program point L with the notation $x_{@L}$. Program points are introduced with the keyword label, in a way similar to the goto labels. A particular label 0 is automatically introduced at the beginning of the program, so that $x_{@0}$ refers to the initial value of x — within a function, that initial value stands for the value at the beginning of the function body i.e. right after the function call. In the post-condition, the keyword *result* stands for the value returned by the computation.

Invariants may be inserted in loops for convenience. Termination of loops and recursive functions is justified by a *variant*, which is the pair of an arbitrary expression ϕ and a relation R over the type of that expression. The relation $\lambda x, y.0 \leq x < y$ is the default relation for variants of type Z. Consequently, the final syntax of loops is the following:

$$\textsf{while } e \textsf{ do } \{ \text{ [}\textsf{invariant } I\text{] } \textsf{variant } \phi \text{ [}\textsf{for } R\text{] } \} \; e \textsf{ done}$$

The tactic to establish the correctness and termination of programs, called Correctness, takes an annotated program, with possible inner annotations, and produces a set of *proof obligations*, which are standard goals in the Coq system. We will not enter the theoretical details of that tactic, which are described in [2] and also in the Coq Reference Manual [1]. The only thing to say is that proof obligations are purely logical propositions, applied to logical variables representing the various values of the mutable variables of the program.

Arrays are indexed over type Z, starting from 0. They are represented in the logical world by an abstract dependent type $(array\ N\ T)$, where N is the size of the array and T the type of its elements. This abstract type is manipulated through the following two functions:

$$\begin{array}{lcl} access & : & \forall N.\, \forall T.\, (array\ N\ T) \rightarrow Z \rightarrow T \\ store & : & \forall N.\, \forall T.\, (array\ N\ T) \rightarrow Z \rightarrow T \rightarrow (array\ N\ T) \end{array}$$

For convenience, we still write $t[i]$ for $(access\ t\ i)$ in specifications. The above two functions are axiomatized as follows:

$$\begin{array}{lcl} store\_def\_1 & : & \forall N.\, \forall T.\, \forall t : (array\ N\ T).\, \forall v : T.\, \forall i : Z. \\ & & \quad 0 \leq i < N \Rightarrow (access\ (store\ t\ i\ v)\ i) = v \\ store\_def\_2 & : & \forall N.\, \forall T.\, \forall t : (array\ N\ T).\, \forall v : T.\, \forall i, j : Z. \\ & & \quad 0 \leq i < N \ \wedge\ 0 \leq j < N \ \wedge\ i \neq j \Rightarrow \\ & & \quad\quad (access\ (store\ t\ i\ v)\ j) = (access\ t\ j) \end{array}$$

The tactic for proving correctness and termination of imperative programs is fully described in the Coq Reference Manual [1], chapter 18. The syntax and the set of available libraries are described, and a bunch of examples demonstrates the use of the tactic.

3 Specifying sorting algorithms

We will only consider here sorting algorithms applying to arrays and working in-place i.e. by moving elements in a single array given as argument. In the following,

N will always denote the size of that array. To simplify the presentation, we limit our case studies to arrays of integers but, of course, all the following still holds for any type with a decidable order relation. When specifying a sorting program, one has to express two facts:

1. first, obviously, that the array is sorted after the evaluation of the program;
2. but also that the values contained in the initial array are *preserved* by the algorithm, which possibly permutes them but does not modify any of them.

Indeed, if we forget the second condition, the following program obviously leads to a sorted array:

$$\mathsf{for}\ i = 0\ \mathsf{to}\ N-1\ \mathsf{do}\ t[i] := 0\ \mathsf{done}$$

but it is surely not what we call a sorting algorithm! In the next two sections, we introduce the specification material corresponding to the above two properties.

3.1 The property of being sorted

We first define a predicate $(sorted\ t\ i\ j)$ which expresses that an array t is sorted in *increasing order* between the bounds i and j. It is formally defined as follows:

$$(sorted\ t\ i\ j) \quad \stackrel{\text{def}}{=} \quad i \leq j \ \wedge\ \forall x.\ i \leq x < j \Rightarrow t[x] \leq t[x+1]$$

Returning to that definition each time you want to establish that a sub-part of an array is sorted is somewhat tedious, and that is why the predicate *sorted* comes with a set of useful lemmas about sorted arrays.

Some of them state rather trivial properties, such as "a sub-part of a sorted part is also sorted" or "any modification of an array outside a sorted part leaves that part sorted", etc. But the most useful lemmas are the ones allowing to increase the sorted part, either on the left side or on the right side of the segment. They can be expressed by the following rules:

$$\frac{(sorted\ t\ i\ j) \quad j < N-1 \quad t[j] \leq t[j+1]}{(sorted\ t\ i\ (j+1))}$$

and

$$\frac{0 < i \quad t[i-1] \leq t[i] \quad (sorted\ t\ i\ j)}{(sorted\ t\ (i-1)\ j)}$$

3.2 Permutation

To express that an array is preserved by a program, we introduce a predicate $(permut\ t\ t')$ whose meaning is that t and t' are permutations of each other. There are many ways to define such a predicate. One possibility would be to state the existence of a one-to-one mapping between the elements of t and the ones of t'. But it would necessitate to exhibit such a mapping each time we want to establish $(permut\ t\ t')$. A better solution is to express that the set of permutations is the smallest equivalence relation containing the transpositions i.e. the exchanges of two elements. So we first define the predicate $(exchange\ t\ t'\ i\ j)$, for two arrays t and t' of type $(array\ N\ Z)$ and two indexes i and j, as follows:

$$\begin{array}{lcl} (exchange\ t\ t'\ i\ j) & \stackrel{\text{def}}{=} & 0 \leq i < N \wedge 0 \leq j < N \wedge t[i] = t'[j] \wedge t[j] = t'[i] \\ & & \wedge\ \forall k.\ 0 \leq k < N \wedge k \neq i \wedge k \neq j \Rightarrow t[k] = t'[k] \end{array}$$

Then we define the predicate $(permut\ t\ t')$ inductively as follows:

$$\frac{(exchange\ t\ t'\ i\ j)}{(permut\ t\ t')} \qquad \frac{}{(permut\ t\ t)}$$

$$\frac{(permut\ t'\ t)}{(permut\ t\ t')} \qquad \frac{(permut\ t\ t') \quad (permut\ t'\ t'')}{(permut\ t\ t'')}$$

Such a solution is very useful in practice, since we will often use the reflexivity — when we do not modify the array — and the transitivity — when we sequence operations that establish permutations. Moreover, two of our three case studies proceed by exchanging elements, namely *quicksort* and *heapsort*, and the proof obligations concerning the permutations are therefore always immediate.

In practice, we also need a predicate to express that a *subpart* $[l..r]$ of two arrays has been permutated, the other elements being untouched. (It is used in *quicksort*, when we apply recursively to subparts of the array.) The corresponding predicate $(sub\_permut\ t\ t'\ l\ r)$ is defined in a way similar to *permut*.

4 Three case studies

Here we come to the three case studies themselves, namely *insertion sort*, *quicksort* and *heapsort*. There are many articles and monographs about sorting algorithms. We mainly refer to two of them. The first one is the excellent book of R. Sedgewick *Algorithms* [6] — a very good introduction to algorithmic and in particular to sorting algorithms — from which we borrowed some pieces of code. The other one is the third volume of D. E. Knuth's *Art of Computer Programming* [4] dedicated to searching and sorting algorithms, which contains a very detailed study of many sorting algorithms, including the three ones we are considering here.

4.1 Insertion sort

The first case study we present is the *insertion sort*. This is one of the basic sorting algorithm taught to computer science students and surely the most natural one. *Insertion sort* is fully described in R. Sedgewick's book [6], pages 98–103, and in D. E. Knuth's one [4], pages 80–102.

The version of *insertion sort* we implement is the simplest one, that traverses the array from left to one, building a sorted array on the left side, and inserts successively the next element in that sorted segment. Thus it is made of two parts, a procedure *insertion* which does the insertion and a main program *insertion_sort* that inserts all the elements one by one.

The procedure *insertion*. That procedure is intended to insert the n-th element of an array t in the already sorted sub-array $t[0..n-1]$. It means that, given the pre-condition

$$(\mathit{sorted}\ t\ 0\ (n-1)) \ \wedge\ 1 \leq n < N$$

the execution of the procedure will establish the following post-condition

$$(\mathit{sorted}\ t\ 0\ n) \ \wedge\ (\mathit{permut}\ t\ t@)$$

We are now going to check how it works in details. The first program one has in mind looks like

```
let v = t[n] in
let j = ref n in
while !j > 0 and t[!j − 1] > v do
  t[!j] := t[!j − 1]; j :=!j − 1
done;
t[!j] := v
```

But such a test in the loop requires a lazy evaluation of the boolean connective and which returns *false* as soon as its first argument is evaluated to *false*, without evaluating its second argument. Otherwise, the array may be accessed outside of its bounds, at index -1. The connectives of the programs considered in the Coq system are strict and if we try to establish the correctness of the above program, we get a proof obligation $0 \leq j-1 < N$ expressing that $t[!j-1]$ is a legal access inside t, which we are not able to establish. Therefore, we rewrite the test using a conditional, in the following way:

```
while (if not (!j > 0) then false else t[!j − 1] > v) do ...
```

From a semantic point of view, the reader may notice that it is exactly a lazy connective. The final code of the procedure *insertion* is given in Figure 1.

The next problem is to find the right invariant for the loop. It should express that the element to insert moves through the array until it reaches its right place. It requires to know, if the current position is j, what are the values at indexes $j-1$ and $j+1$. Since those indexes may be outside the array, we cannot define a unique general invariant. Therefore, we introduce a case-based invariant, by distinguishing the three cases $j = 0$, $0 < j < n$ and $j = n$.

The general case $0 < j < n$ states that the property of being sorted holds in the two sub-arrays located on each side of the current index j as illustrated below, with the two additional properties $t[j-1] \leq t[j+1]$ and $v \leq t[j]$.

0	j	$n+1$ $\quad$ $N-1$
sorted	sorted	

```
fun (N : Z)(t : array N of Z)(n : Z) →
 { (sorted_array t 0 (n − 1)) ∧ 1 ≤ n < N }
 (let v = t[n] in
  let j = ref n in
  begin
    while
      (if not (!j > 0) then false else t[!j − 1] > v)
      { if result then j > 0 ∧ (access t (j − 1)) > v
                  else j = 0 ∨ (j > 0 ∧ (access t (j − 1)) ≤ v) }
    do
      { invariant ( (global_inv N t j n v)
                  ∧ (inv_0 N t j n v)
                  ∧ (inv_n N t j n v) )
                ∧ (permut (store t j v) t@0)
                ∧ 0 ≤ j ≤ n
        variant j }
        t[!j] := t[!j − 1];
        j := !j − 1
    done;
    t[!j] := v
  end)
  { (sorted_array t 0 n) ∧ (permut t t@) }
```

Fig. 1. The procedure *insertion*

The other two cases, when $j = 0$ and when $j = n$, and degenerated forms of that invariant. Hence our invariant will contain the conjunction of three predicates, which are the following:

$$
\begin{array}{lcl}
(inv\_n\ t\ j\ n\ v) & \stackrel{\text{def}}{=} & j = n \Rightarrow (sorted\ t\ 0\ (j-1))\ \wedge\ v \leq t[j] \\
(inv\_0\ t\ j\ n\ v) & \stackrel{\text{def}}{=} & j = 0 \Rightarrow (sorted\ t\ j\ n)\ \wedge\ v \leq t[j] \\
(global\_inv\ t\ j\ n\ v) & \stackrel{\text{def}}{=} & 0 < j < n\ \Rightarrow \\
 & & (sorted\ t\ 0\ (j-1))\ \wedge\ (sorted\ t\ j\ n) \\
 & & \wedge\ t[j-1] \leq t[j+1]\ \wedge\ v \leq t[j]
\end{array}
$$

The last property to establish is the preservation of the elements of the array. There is a slight difficulty here, since during the loop iterations the array t is no more a permutation of the initial array. Indeed, an element is duplicated in $t[j]$ and $t[j-1]$, while the element to insert is kept in the variable v. Therefore, we do not express that t is a permutation of the initial array $t_{@0}$ but that $(store\ t\ j\ v)$ is a permutation of $t_{@0}$.

The sorting program itself. Once the proof of the procedure *insertion* is completed, it is quite easy to certify the main program *insertion_sort*. It is simply a loop which calls the procedure insertion for each position from 1 to the right bound of the array. The invariant is simply:

$$(sorted\ t\ 0\ (i-1))\ \wedge\ (permut\ t\ t_{@0})\ \wedge\ 1 \leq i$$

which is easily established by the post-condition of *insertion*. The annotated code of *insertion_sort* is given in Figure 2.

```
fun (N : Z)(t : array N of Z) →
  { N > 0 }
  let i = ref 1 in
    begin
      while !i < N do
        { invariant (sorted_array t 0 (i − 1)) ∧ (permut t t@0) ∧ (1 ≤ i)
          variant N − i }
        (insertion N t !i);
        i := !i + 1
      done
    end
    { (sorted_array t 0 (N − 1)) ∧ (permut t t@0) }
```

Fig. 2. The program *insertion_sort*

4.2 Quicksort

It is not necessary to introduce the *quicksort* algorithm, since it is the most widely taught algorithm, as the paradigm of the divide-and-conquer methodology, and also the most studied one, as one of the most efficient sorting algorithm. *Quicksort* is due to C. A. R. Hoare [3] and has been improved and studied by R. Sedgewick in his Ph.D.'s thesis [5]; the pages 115–131 of his book [6] are devoted to it. A detailed study is available in D. E. Knuth's book [4], pages 114–123.

We are going to write *quicksort* in the usual way, with two distinct parts: first a *partitioning* function, which takes a sub-part of the array as argument, rearranges elements in that segment to obtain a left part with values less or equal than those of the right part, and returns the position of the partitioning element; secondly, a recursive *divide-and-conquer* procedure which call the partitioning function and then calls itself recursively on both left and right sides of the partitioning element.

Partitioning. *Quicksort* belongs to the family of sorting algorithms which proceed by *exchanging* elements of the array. Thus it is a good idea, for the clarity of the code, to introduce a procedure *swap* which exchanges in the array t the two elements at positions i and j. The post-condition of that procedure will be simply

$$(exchange\ t\ t_@\ i\ j) \tag{1}$$

where the predicate *exchange* has been introduced in Section 3. The code of *swap* is given in Figure 3, and its proof of correctness is immediate. Notice that it is not necessary to express that t is preserved by the procedure *swap* since it is a consequence of (1) by definition of the predicate *permut*.

```
fun (N : Z)(t : array N of Z)(i, j : Z) →
 { 0 ≤ i < N ∧ 0 ≤ j < N }
 let v = t[i] in begin t[i] := t[j]; t[j] := v end
 { (exchange t t@ i j) }
```

Fig. 3. The procedure *swap*

Partitioning the segment $[l..r]$ of the array t consists in selecting a value v in $t[l..r]$ and re-arranging the elements of that segment to obtain the following situation:

	0	l	p		r $\quad N-1$
t		$\leq v$	v	$\geq v$	

We are going to write a partitioning function *partition* which establishes the above situation and returns the index p, called the partitioning element. To specify that function, we define a predicate $(partition\_p\ t\ l\ r\ p)$ which expresses the above property. It is the conjunction of three properties, the first one expressing that p belongs to the interval $[l..r]$, and the other two ones expressing that the values of the left and right sides of p are respectively less or equal and greater or equal to $v = t[p]$. The definitions are the following:

$$\begin{array}{lcl}
(array_le\ t\ l\ r\ v) & \stackrel{\text{def}}{=} & \forall i.\ g \leq i \leq d \Rightarrow t[i] \leq v \\
(array_ge\ t\ l\ r\ v) & \stackrel{\text{def}}{=} & \forall i.\ g \leq i \leq d \Rightarrow v \leq t[i] \\
(partition_p\ t\ l\ r\ p) & \stackrel{\text{def}}{=} & g \leq p \leq d\ \wedge (array_le\ t\ l\ (p-1)\ t[p]) \\
 & & \qquad\qquad \wedge (array_ge\ t\ (p+1)\ r\ t[p])
\end{array}$$

Then, the specification of the function *partition* is quite simple: its pre-condition expresses that the bounds l and r are such that $0 \leq l < r < N$ holds, and its post-condition expresses that the sub-part $t[l..r]$ has been re-arranged in such a way that the returned value — which is denoted by *result* — is a partitioning element for the segment $[l..r]$ i.e.

$$(partition\_p\ t\ l\ r\ result) \wedge (sub\_permut\ l\ r\ t\ t_{@})$$

Actually, it is a good idea to add explicitly the property $l \leq result \leq r$ in the post-condition, even if it is already included in the definition of *partition_p*.

Regarding the code, we didn't make a complex choice of the partitioning element, as it is often the case in efficient implementations of *quicksort*. We simply chose the first element $t[l]$, as in academic presentations. However, a more complex choice wouldn't change the specification, and would only make some proof obligations a bit more complicated — and more numerous since the code would also be bigger. The code of the partitioning function *partition* is given in Figure 4. It is written in a standard way, with two indexes i and j scanning the segment $[l..r]$ respectively from left to right and right to left, exchanging the values $t[i]$ and $t[j]$ when they are on the wrong side of the partitioning value, until they cross each other, which gives the final position of the partitioning element — one has to take care at that point, since the partitioning position may be i or $i-1$. The invariants of the three loops are self-explanatory. The label L is used to express that the indexes i and j keep respectively increasing and decreasing, which is necessary to establish to termination of the outer loop.

Divide-and-conquer. The second part of the *quicksort* algorithm applies the famous technique known as *divide-and-conquer*: to sort the segment $[l..r]$ of the array t, we call the function *partition*, which establishes the partition and returns the position p of the partitioning element, and we call recursively the same program on both segments $[l, p-1]$ and $[p+1, r]$, until the segments contain at most one element. Therefore, the recursive function *quick_rec* which implements this algorithm has the pre-condition $0 \leq l \wedge r < N$ and the following post-condition:

$$(sorted\_array\ t\ l\ r) \wedge (sub\_permut\ l\ r\ t\ t_{@})$$

which expresses that the segment $t[l..r]$ has been sorted and that we only proceeded by a permutation of the elements of that part of t. The code for that function is very simple to write, and is given in Figure 5. Its termination is easy to prove, using the width of the segment $[l..r]$ as variant, that is $r - l$, since it clearly decreases in each recursive call. That code leads to a few number of proof obligations, among which one is non trivial. It corresponds to the establishment of the post-condition after the three function calls and can be illustrated by the following situation:

```
fun (N : Z)(t : array N of Z)(l, r : Z)  →
  {  0 ≤ l < r  ∧  r < N  }
 (let pv  =  t[l] in
  let i  =  ref (l + 1) in
  let j  =  ref r in
  begin
    while !i < !j do
      {  invariant l + 1 ≤ i ≤ r  ∧  j ≤ r
              ∧  (array_le t (l + 1) (i − 1) pv)  ∧  (array_ge t (j + 1) r pv)
              ∧  (sub_permut l r t t@0)  ∧  t[l] = t@0[l]
        variant j − i for (Zwf (−N − 2))  }
      label L;
      while t[!i] ≤ pv and !i < !j do
        {  invariant i@L ≤ i ≤ r  ∧  (array_le t (l + 1) (i − 1) pv)
          variant r − i  }
        i := !i + 1
      done;
      while t[!j] ≥ pv and !i < !j do
        {  invariant l ≤ j ≤ j@L  ∧  (array_ge t (j + 1) r pv)
          variant j  }
        j := !j − 1
      done;
      if !i < !j then begin
        (swap N t !i !j);
        i := !i + 1;
        j := !j − 1
      end
    done;
    if t[!i] < pv then begin
      (swap N t l !i);
      !i
    end else begin
      (swap N t l (!i − 1));
      !i − 1
    end
  end)
  {  l ≤ result ≤ r  ∧  (partition_p t l r result)  ∧  (sub_permut l r t t@)  }
```

Fig. 4. The function *partition*

```
let rec quick_rec (N : Z)(t : array N of Z)(l, r : Z)  :  unit
{  variant r − l for (Zwf (−1))  }  =
  {  0 ≤ l ∧ r < N   }
  (if l < r then
     let p  =  (partition N t l r) in
     begin
       (quick_rec N t l (p − 1));
       (quick_rec N t (p + 1) r)
     end)
  {  (sorted_array t l r) ∧ (sub_permut l r t t@0)  }.
```

Fig. 5. The recursive function *quick_rec*

	0	l	p		r $\quad N-1$
t_0					
t_1	$= t_0$	$\leq v$	v	$\geq v$	$= t_0$
t_2	$= t_1$	sorted	v	$= t_1$	$= t_1$
t_3	$= t_2$	$= t_2$	v	sorted	$= t_2$

where t_0 is the initial value of t, t_1 its value after the call to the partitioning function, and t_2 and t_3 its values after the two recursive calls. This proof obligations is established as a lemma, whose formal statement is the following:

$$\frac{\begin{array}{l} \forall t_0, t_1, t_2, t_3.\ \forall l, r, p. \\ 0 \leq l < r < N \\ (partition\_p\ t_1\ l\ r\ p)\ \wedge\ (sub\_permut\ l\ r\ t_1\ t_0) \\ (sorted\_array\ t_2\ l\ (p-1))\ \wedge\ (sub\_permut\ l\ (p-1)\ t_2\ t_1) \\ (sorted\_array\ t_3\ (p+1)\ r)\ \wedge\ (sub\_permut\ (p+1)\ r\ t_3\ t_2) \end{array}}{(sorted\_array\ t_3\ l\ r)\ \wedge\ (sub\_permut\ l\ r\ t_3\ t_0)}$$

At last, the sorting program itself is just a call to *quick_rec* on the whole array i.e. with the bounds 0 and $N - 1$. The code is given in Figure 6 and its proof of correctness is immediate.

4.3 Heapsort

We kept the best last. Indeed, even if *quicksort* is preferred to *heapsort* in practice, the latter has the best complexity in worst case, namely $O(n \log n)$, which is the optimal complexity for a sorting algorithm. Moreover, *heapsort* is a very beautiful algorithm, based on a shrewd data structure, and which can be implemented in-place with only a few lines of code. As for the two other sorting algorithms, the reader can find a description of *heapsort* in R. Sedgewick's book [6],

```
fun (N : Z)(t : array N of Z) →
  (quick_rec N t 0 (N − 1))
  { (sorted_array t 0 (N − 1)) ∧ (permut t t@) }
```

Fig. 6. The program *quicksort*

pages 153–158, and a detailed study in *The Art of Computer Programming* [4], pages 149–153.

The *heapsort* algorithm uses the structure of *heap*. Heaps, which are sometimes called *tournament trees*, are complete binary trees where each node has a value greater or equal to those of its two sons — or that of its unique son. Heaps are mainly used to implement priority queues. Indeed, in such a structure, getting the greatest value is immediate — it is the root of the tree — and adding or removing an element can be done in logarithmic time with respect to the number of elements. The *heapsort* implementation uses the fact that a binary tree can be represented inside an array where only the values of the nodes are stored, and where the two sons of a node stored an index i are stored at indexes $2i+1$ and $2i+2$. Assuming that we know how to manipulate the heap with such a representation, the sorting algorithm proceeds as follows:

1. first, it builds a heap with all the elements of the array, in the array itself;
2. secondly, it sorts the array progressively by keeping a heap on the left part of the array and a set of already sorted elements on the right part, which can be illustrated like this:

	0	$k+1$ $\quad N-1$
t	heap with elements $\leq t[k+1]$	part already sorted

We used here the code of R. Sedgewick ([6] page 155), which is incredibly short and clever. To specify that program, we have to give a formal definition to the heap structure.

Definition of the *heap* structure. We define a predicate $(heap\ t\ n\ k)$ whose meaning is the following: "In the array t, the tree of root k made of elements at indexes less or equal than n is a heap." That predicate *heap* is inductively defined as follows:

$$\frac{\begin{array}{c}2k+1 \leq n \Rightarrow (t[k] \geq t[2k+1] \wedge (heap\ t\ n\ (2k+1))) \\ 2k+2 \leq n \Rightarrow (t[k] \geq t[2k+2] \wedge (heap\ t\ n\ (2k+2)))\end{array}}{(heap\ t\ n\ k)}$$

Even if most lemmas about heaps will be proved using the induction principle associated to the above definition, a few lemmas will necessitate another induction principle, on natural numbers, which follows the encoding of heaps. That principle is the following:

$$\begin{array}{l}\forall P : Z \rightarrow \mathsf{Prop}.\\ \quad (P\ 0) \Rightarrow\\ \quad (\forall x : Z.\ 0 \leq x \Rightarrow (P\ x) \Rightarrow (P\ (2x+1)) \wedge (P\ (2x+2))) \Rightarrow\\ \quad \forall x : Z.\ 0 \leq x \Rightarrow (P\ x)\end{array}$$

Another predicate will be useful to specify the code of *heapsort*. It expresses that all the elements of a tree — still represented as above but which is not necessarily a heap — are smaller or equal than a given value v. That predicate, written $(\mathit{inftree}\ t\ n\ v\ k)$, is also inductively defined, as follows:

$$\frac{\begin{array}{c} t[k] \leq v \\ 2k+1 \leq n \Rightarrow (\mathit{inftree}\ t\ n\ v\ (2k+1)) \\ 2k+2 \leq n \Rightarrow (\mathit{inftree}\ t\ n\ v\ (2k+2)) \end{array}}{(\mathit{inftree}\ t\ n\ v\ k)}$$

Building and destructuring the heap. The code of *heapsort* is usually made of two parts, one to build the heap with the use of a function *upheap*, and one to sort the elements, removing them from the heap with the use of a function *downheap*. R. Sedgewick shows in [6] that both parts may be written using only the procedure *downheap*, which leads to a very concise and elegant implementation.

The procedure *downheap* takes as argument an array t, an index k representing the root of a tree, and a maximal index n, with the pre-condition that all the elements of the tree rooted at k, of indexes less or equal than n, form a heap *except may be the root k itself*. Then it moves the value in $t[k]$ down in the tree, by exchanging it with the greatest of its two sons, until the property of heap holds for the whole tree rooted at k. In our case, we can lower the generality of this function and give it the following specification:

$$\begin{array}{l}\{\ \forall i : Z.\ k+1 \leq i \leq n \Rightarrow (\mathit{heap}\ t\ n\ i)\ \}\\ (\mathit{downheap}\ t\ k\ n)\\ \{\ (\mathit{permut}\ t\ t_{@})\ \wedge\ \forall i : Z.\ k \leq i \leq n \Rightarrow (\mathit{heap}\ t\ n\ i)\ \}\end{array} \tag{2}$$

which expresses that, if we have heaps at every position between $k+1$ and n then, after the call of $(\mathit{downheap}\ t\ k\ n)$, we will have heaps at every position between k and n. (We also express the preservation of the array, which will be needed to establish its preservation by the main program.) Assuming that *downheap* is available, *heapsort* is really easy to write. To build the heap, the trick consists in using a *bottom-up* approach, which can be done in only one line of code:

$$\mathsf{for}\ k = (N-2)/2\ \mathsf{downto}\ 0\ \mathsf{do}\ (\mathit{downheap}\ t\ k\ (N-1))\ \mathsf{done}$$

Then, sorting the array can be done with another line of code, by successively exchanging the greatest element of the heap — which is at position 0 — with the last element of the heap:

$$\mathsf{for}\ k = N-1\ \mathsf{downto}\ 1\ \mathsf{do}\ (\mathit{swap}\ t\ 0\ k);\ (\mathit{downheap}\ t\ 0\ (k-1))\ \mathsf{done}$$

This leads to the code given in Figure 7, where the invariants were easy to find. (The procedure *swap* has already been introduced in section 4.2.)

```
fun (N : Z)(t : array N of Z) →
 { 1 ≤ N }
 begin
  for k = (N − 2)/2 downto 0 do
   { invariant −1 ≤ k ≤ N − 1
          ∧ (∀i : Z. k + 1 ≤ i ≤ N − 1 ⇒ (heap t (N − 1) i))
          ∧ (permut t t@0)
     variant k + 1 }
   (downheap N t k (N − 1))
  done;
  assert { (heap t (N − 1) 0) ∧ (permut t t@0) };
  for k = N − 1 downto 1 do
   { invariant 0 ≤ k ≤ N − 1
          ∧ (∀i : Z. 0 ≤ i ≤ k ⇒ (heap t k i))
          ∧ (k + 1 ≤ N − 1 ⇒ t[0] ≤ t[k + 1])
          ∧ (k + 1 ≤ N − 1 ⇒ (sorted_array t (k + 1) (N − 1)))
          ∧ (permut t t@0)
     variant k }
   (swap N t 0 k);
   (downheap N t 0 (k − 1))
  done
 end
 { (sorted_array t 0 (N − 1)) ∧ (permut t t@) }
```

Fig. 7. The program *heapsort*

The procedure *downheap*. We still have to implement the *downheap* function. It is usually written in an imperative style, with some code looking like[1]:

```
downheap t k0 n ≝
 let k = ref k0 in
 while 2×!k + 1 ≤ n do
  let j = [ select the greatest son of k ] in
  if t[j] ≤ t[!k] then break;
  (swap t !k j);
  k := j
 done
```

[1] The code of R. Sedgewick does not use a break but a goto, but with the same purpose.

Since we do not have a `break` construction in our language, we have to write it differently, keeping the same idea of stopping the process as soon as the node k is greater or equal to its son(s). A good idea is to write it as a recursive function, in the following way:

$$
\begin{array}{l}
\textit{downheap}\ t\ k\ n\ \stackrel{\text{def}}{=} \\
\quad \mathsf{if}\ 2 \times k + 1 \leq n\ \mathsf{then} \\
\qquad \mathsf{let}\ j = [\ \textit{select the greatest son of}\ k\ ]\ \mathsf{in} \\
\qquad \mathsf{if}\ t[k] < t[j]\ \mathsf{then\ begin} \\
\qquad\quad (\textit{swap}\ t\ k\ j); \\
\qquad\quad (\textit{downheap}\ t\ j\ n) \\
\qquad \mathsf{end}
\end{array}
$$

The code to select the greatest son of k, namely j, takes care of the case when k has only one son. That piece of code is "abstracted" by being given the post-condition $(\textit{select\_son}\ t\ k\ n\ j)$ which is defined as:

$$
\begin{array}{lcl}
(\textit{select\_son}\ t\ k\ n\ j) & \stackrel{\text{def}}{=} & \quad (j = 2k+1\ \wedge\ (2k+2 \leq n \Rightarrow t[j] \geq t[2k+2])) \\
 & & \vee\ (j = 2k+2\ \wedge\ j \leq n\ \wedge\ t[j] \geq t[2k+1])
\end{array}
$$

When we come to the proof of correctness of the procedure *downheap*, with the specification (2), we discover that we are not able to establish all the proof obligations. The reason is that our specification of *downheap* is not strong enough; in particular, we didn't state what our function *does not do*. Indeed, after the recursive call $(\textit{downheap}\ t\ j\ n)$ we get an array with the heap property at root j, but we do not know if some elements outside that tree have been modified, and we are not able to establish the heap property at root k. Consequently, we have to strengthen the post-condition. First, we claim that some elements are not touched by the procedure, namely those with indexes i such that $i < k$, $k < i < 2k+1$ or $n < i$. Secondly, we claim that the maximal value of the tree at root k didn't increase, which is expressed by:

$$\forall v.\ (\textit{inftree}\ t_{@}\ n\ v\ k) \Rightarrow (\textit{inftree}\ t\ n\ v\ k)$$

Then, with such a post-condition, the proof obligations can be established. The final annotated code of *downheap* is given in Figure 8.

5 Discussion

Those three case studies show that the Coq system can be an effective framework for developing proofs of pretty complex imperative programs. The tactic used leads to very natural proof obligations, which are very close to the lemmas one would require in a hand-made proof using Floyd-Hoare logic.

Some significant figures about the developments we presented are put together in Figure 9. In particular, they show that the full development of a formal

```
let rec downheap (N : Z)(t : array N of Z)(k, n : Z) : unit { variant n − k } =
  { 0 ≤ k ≤ n ∧ n < N
    ∧ ∀i : Z. k + 1 ≤ i ≤ n ⇒ (heap t n i) }
  (let j = 2 × k + 1 in
   if j ≤ n then
     let j′ = (if j + 1 ≤ n then if t[j] < t[j + 1] then j + 1 else j else j)
              { (select_son t k n result) } in
     if t[k] < t[j′] then begin (swap N t k j′); (downheap N t j′ n) end)
  { (permut t t@)
    ∧ (∀i : Z. k ≤ i ≤ n ⇒ (heap t n i))
    ∧ (∀i : Z. (0 ≤ i < k ∨ k < i < 2 × k + 1 ∨ n < i < N) ⇒ t[i] = t@[i])
    ∧ (∀v : Z. (inftree t@ n v k) ⇒ (inftree t n v k)) }
```

Fig. 8. The recursive procedure *downheap*

proof requires a reasonable amount of time (a few days). The total development time for *insertion sort* is not given, since it involved the development of the libraries and also the debugging of the correctness tool. The annotations and the proof obligations are not so numerous, in the same order of magnitude than the lines of code. Moreover, the libraries we developed, about sorted arrays and permutations, are independent of the programs and can be reused in other proofs, allowing faster and easier developments.

Although it sometimes lacks more powerful automatic tactics, especially to solve trivial goals, the Coq system provides some very useful tactics to solve proof obligations without too much interacting with the system. The tactic Omega, which is intended to solve quantifier-free problems in the Presburger arithmetic, is of great help to the user when he or she tries to solve goals about the decreasing of the variant or the inequalities expressing legal accesses within the bounds of arrays.

But the main advantage of those developments is surely to have demonstrated the relevance of a powerful logic, namely the Calculus of Inductive Constructions, in the process of specifying programs and proving their correctness. Indeed, we have defined and used some inductive predicates, as for instance *permut* and *heap*, and even used higher order to prove a new induction principle based on the coding of heaps.

Acknowledgements. The authors wish to thank Christine Paulin and Benjamin Monate for their help during the process of specifying and proving some parts of the programs.

References

[1] The Coq Proof Assistant. http://coq.inria.fr/.

	insertion sort	quicksort	heapsort
lines of specification	15	13	23
lines of code	17	41 (4)	19 (2)
number of annotations	7	18	7
number of proof obligations	14	26	22
lemmas (manually introduced)	8	11	28
proof steps	300	468	626
total development time	—	2 days	3 days

Fig. 9. Some significant figures

[2] J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. In *Proceedings of the TYPES'98 workshop*, 1998. To appear.

[3] C. A. R. Hoare. Quicksort. *Computer Journal*, 1(5), 1962.

[4] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[5] R. Sedgewick. *Quicksort.* Garland, New York, 1978. Also appeared as the author's Ph.D. dissertation, Stanford University, 1975.

[6] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.