Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 332–340
Science of Computer Programming
www.elsevier.com/locate/scico

Formal proof of a program: Find

Jean-Christophe Filliâtre

LRI–CNRS, Université Paris Sud, 91405 Orsay, France

Received 4 February 2001; received in revised form 31 August 2001; accepted 11 October 2006
Available online 28 November 2006

Abstract

In 1971, C.A.R. Hoare gave the proof of correctness and termination of a rather complex algorithm, in a paper entitled *Proof of a program: Find*. It is a handmade proof, where the program is given together with its formal specification and where each step is fully justified by mathematical reasoning. We present here a formal proof of the same program in the system Coq, using the recent tactic of the system developed to establish the total correctness of imperative programs. We follow Hoare's paper as closely as possible, keeping the same program and the same specification. We show that we get exactly the same proof obligations, which are proved in a straightforward way, following the original paper. We also explain how more informal aspects of Hoare's proof are formalized in the system Coq. This demonstrates the adequacy of the system Coq in the process of certifying imperative programs.

Keywords: Formal methods; Imperative programs; Hoare logic; Coq proof assistant

1. Introduction

"Computer programming is an exact science" is the main assertion of C.A.R. Hoare's famous paper *An axiomatic basis for computer programming* [1]. Even without any tool support, the process of writing software may be completely formal, and mathematical reasoning can be applied to justify every step. A few years later, Hoare clearly demonstrated this assertion by publishing the fully detailed proof of a quite complex algorithm, *Find* [2]. This algorithm was published ten years before by Hoare himself [3], but is better known as *Select* in the algorithmic literature. It consists, given an array A of comparable elements and a particular index f within this array, in reorganizing the elements of A in such a way that all the elements on the left side of index f are smaller than or equal to $A[f]$ and all the elements on the right side of index f are greater than or equal to $A[f]$. Such an algorithm is useful, for instance, for finding the median of a set of elements without sorting them.

Hoare's proof is based on the use of invariants. Proof obligations are made explicit at each step, and proved as lemmas. Hoare shows how the program can be built in a top-down way, proof obligations being extracted and proved as soon as possible. He first considers the proof of correctness, and then the proof of termination is done in a next section. The preservation of the array's elements is the subject of a separate proof. Finally, Hoare notices that proving that all indices used are within the bounds of the array would be necessary, but does not do it.

E-mail address: filliatr@lri.fr.
URL: www.lri.fr/~filliatr.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.10.002

Hoare's proof is quite impressive. The reader can appreciate the mathematical rigor of the development. Although it is done without any tool support, it illustrates a very precise methodology and incorporates modern ideas like refinement. When reading his paper, one immediately imagines how it would be nice to have a tool to support such program proofs. Hoare's logic [1] and related works of the early seventies have actually been waiting for tool support for years, even decades. The main reason was not the methods themselves but relates instead to the lack of formal logical frameworks to support them. There are nowadays many such frameworks, and software certification may at last be mechanically assisted.

In the author's thesis [4–6], a method for establishing the total correctness of programs mixing functional and imperative features is introduced, in the framework of Type Theory. The main idea is to build, for a given annotated program $\{P\}\ e\ \{Q\}$, a proof of the property $\forall \vec{x}.\ P(\vec{x}) \Rightarrow \exists \vec{y}.\ Q(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ respectively stand for the input and output variables of the program e. This proof has an informative content which respects the semantics of e, and missing parts corresponding to the proof obligations. One of the main advantages of such an approach is a direct treatment of functional constructions, including function calls and recursive functions. This method is implemented in the Coq proof assistant [7] and has been applied to the proof of complex algorithms [5,8].

When we began experimenting with the method, Hoare's proof appeared as a good candidate. First, it was a non-trivial program, although quite academic. Second, it was already specified and proved, so that we could concentrate on the method itself rather than on specifying or proving tasks. We closely followed Hoare's program and specification and, surprisingly, we found *exactly the same proof obligations* as those given in [2]. Then we easily followed Hoare's proofs to discharge the proof obligations. This paper describes this formal development in the system Coq, and details the relationships and differences between Hoare's paper and our fully formal proof. The original constructions of program and proof are not given in this paper, but Hoare's final annotated program is given in Section 3. The reader might refer to [2] for further details.

This article is organized as follows. Section 2 quickly introduces the proof of imperative programs in the system Coq. Then Section 3 details the formal development in the system Coq, describing the various annotations and comparing them to the original ones. In the last section, we give a global comparison of the two proofs and discuss the ease of such a formal development.

2. Certifying programs in the system Coq

The system Coq [7] is a proof assistant for the Calculus of Inductive Constructions, a higher-order extension of Girard's system F with dependent types and inductive predicates, developed by Coquand, Huet and Paulin-Mohring [9, 10]. Within this framework, the present author developed a methodology to establish the total correctness of programs mixing functional and imperative features [4,5].

The programming language includes the usual constructs of imperative programs, namely references and arrays, sequences, conditionals and loops, but also functional constructs, namely possibly recursive functions and procedures, calls by value and by reference, and even restricted kinds of polymorphism and higher-order functions. Base types belong to the underlying logic, that is any inductive type definable in the Calculus of Inductive Constructions. There is no distinction between statements and expressions, and consequently expressions may include side-effects. The syntax is close to the one of ML: a reference x is created with the construct ref, accessed as $!x$, and modified as $x := e$; local variables are introduced with the let in construct, and necessarily initialized; the other constructs are usual.

Programs are annotated with preconditions and postconditions, using the traditional notation of Hoare logic. These assertions are arbitrary propositions over the program's variables. The current value of a reference x is directly referred to as x. In a postcondition, its value before the evaluation (at the precondition point) is referred to as $x_@$. More generally, labels may be inserted inside programs with the keyword label, so that the value of a reference at a given labelled point L can be referred to as $x_{@L}$. A particular label 0 is automatically inserted at the beginning of the program, and therefore $x_{@0}$ stands for the initial value of x. This facility avoids the painful use of auxiliary variables, as illustrated later in this section.

A keyword assert can be used to claim that a proposition is true at a given point inside a sequence. It is useful to establish once a property that will be used several times in later proofs, or just to make the program clearer. A keyword invariant declares a loop invariant. It comes together with a variant introduced with the syntax variant ϕ for R. ϕ is

an arbitrary expression of any type, while R is a binary relation over that type. When R is not given, ϕ must be an integer expression and the relation is $\lambda x, y.\ 0 \leq x < y$. In this paper, we will use the latter kinds of variants only.

Arrays are axiomatized as an abstract datatype (array n T) where n is the size of the array and T the type of its elements. Access and assignment are abstract operations of respective types

$$\begin{aligned}&\mathsf{access} : \forall n : Z.\, \forall T : \mathsf{Set}.\, (\mathsf{array}\ n\ T) \to Z \to T\\ &\mathsf{store}\ \ : \forall n : Z.\, \forall T : \mathsf{Set}.\, (\mathsf{array}\ n\ T) \to Z \to T \to (\mathsf{array}\ n\ T)\end{aligned}$$

where Z is Coq's datatype for integers. In the following, (access t i) is written $t[i]$. Arrays are indexed from 0 to $n-1$ but to keep close to Hoare's original program and proof, we will use arrays indexed from 1 to n, as he did.

The implementation of the method in the system Coq relies on a tactic called Correctness, which takes an annotated program and generates the proof obligations, and on a few commands to declare functions and variables and to inspect the context. The obligations are standard Coq goals (i.e. propositions to prove) and consequently the user can use any Coq tactic to establish their validity. The Correctness tactic works as follows: First, it performs a static effects analysis of the program, which determines the sets of variables accessed and/or modified by each subexpression. Then it builds a functional interpretation of the program using monads; however, it does not use store-based monads as in the traditional approach, but finer grain monads that carry the variables' values separately, according to the effects [5]. Finally, an incomplete proof term is obtained by inserting proof placeholders at the logical places and is fed to the Coq proof engine.

For instance, the following program

$$\mathsf{begin}\ x := !x - 1;\ y := !y - 1\ \mathsf{end}\ \{\, x + y = x_@ + y_@ \,\}$$

modifies both references x and y. Thus it is to be interpreted as a proof of the proposition $\forall x_0, y_0.\, \exists x_1, y_1.\, x_1 + y_1 = x_0 + y_0$. This proof looks like

$$\begin{aligned}&\lambda x_0, y_0.\ \mathsf{let}\ x_1 = x_0 + 1\ \mathsf{in}\\ &\qquad \mathsf{let}\ y_1 = y_0 - 1\ \mathsf{in}\\ &\qquad ((x_1, y_1),\ ? : x_1 + y_1 = x_0 + y_0)\end{aligned}$$

where the two let constructs correspond to the monadic composition operator. They introduce new variables, namely x_1 and y_1, to represent the new values of the references x and y. Proof obligations are propositions over such variables. In this example, there is only one proof obligation, identified by the question mark, to establish the proposition

$$\forall x_0, y_0, x_1, y_1.\, x_1 = x_0 + 1 \wedge y_1 = y_0 - 1 \Rightarrow x_1 + y_1 = x_0 + y_0.$$

The tactic itself is written in Objective Caml, the programming language in which Coq is written, and is approximatively 6000 lines long. It comes with a few libraries, mainly dealing with arrays and their properties. Up to now, no specific tactic was developed to help the user in proving the obligations. The tactic Correctness is already distributed with the system, and documented in the Coq's reference manual [7], where a few examples are given. More complex proofs of correctness are available on the Coq web site [7], including in-place sorting algorithms [8], the Knuth–Pratt–Morris string searching algorithm [5], Floyd's algorithm, Petersson's mutual exclusion algorithm [11], etc.

3. A formal proof

Hoare's program *Find* is given in Fig. 1, exactly as it appears at the end of [2]. *Find* works by a 'pivoting' mechanism similar to that of *Quicksort*. Given an array A of length N and a position f, it operates at each step on what Hoare calls the 'middle section', $A[m] \ldots A[n]$, containing position f, with m and n initially 1 and N respectively. In each iteration, *Find* selects a 'pivot' r from the middle section (given by $A[f]$, in fact, although this choice is immaterial), then permutes the middle section so that entries smaller than r move to its left and larger entries to its right. If r ends up at position f, then *Find* terminates; otherwise, it repeats the process, focusing the middle section on whichever part – above or below r – contains index f. As this removes at least the pivot r, the middle section gets smaller at each step, ensuring termination.

```
1.  begin
2.   integer m, n;
3.   comment { m ≤ f ∧ ∀p, q : Z. 1 ≤ p < m ≤ q ≤ N ⇒ A[p] ≤ A[q],
4.                 f ≤ n ∧ ∀p, q : Z. 1 ≤ p ≤ n < q ≤ N ⇒ A[p] ≤ A[q] };
5.   m := 1; n := N;
6.   while m < n do begin
7.    integer r, i, j, w;
8.    comment { m ≤ i ∧ ∀p : Z. 1 ≤ p < i ⇒ A[p] ≤ r,
9.                 j ≤ n ∧ ∀q : Z. j < q ≤ N ⇒ r ≤ A[q] };
10.   r := A[f]; i := m; j := n;
11.   while i ≤ j do begin
12.    while A[i] < r do i := i + 1;
13.    while r < A[j] do j := j − 1;
14.    comment { A[j] ≤ r ≤ A[i] };
15.    if i ≤ j then begin
16.     w := A[i]; A[i] := A[j]; A[j] := w;
17.     comment { A[i] ≤ r ≤ A[j] };
18.     i := i + 1; j := j − 1
19.    end
20.   end;
21.   if f ≤ j then n := j
22.   else if i ≤ f then m := i
23.   else goto L
24.  end;
25.  L :
26.  end
27.  { ∀p, q : Z. 1 ≤ p ≤ f ≤ q ≤ N ⇒ A[p] ≤ A[f] ≤ A[q] }
```

Fig. 1. The original code and annotations of *Find* (excerpt from [2]).

The annotated code of our formal development[1] is given in Fig. 2, at the end of this paper. The lines are numbered and we use these numbers in the following to refer to parts of the code and its specification (e.g. "see invariants lines 3–4, 9–11, 15–16 and 21–22"). We keep Hoare's terminology to designate the program loops: there are the "main loop" (line 2), the "middle loop" (line 8) and the two "inner loops" (lines 14 and 20).

Our code is exactly the original given in Fig. 1 except for two details. Firstly, since local variables must be initialized when declared, a piece of code like begin integer i; $i := 0$; ... end is translated into let i = ref 0 in ... (lines 1 and 6). Moreover, since the variables r and w need not be mutable, they are introduced with a let instead of a let ref (lines 6 and 28). Secondly, the goto statement at the end of the original code is removed. Its role is to exit the main loop when both i and j have crossed f. In our program, the same effect is achieved by assigning both m and n the same value (namely f), which makes the test of the main loop become false (lines 2 and 40).

When fed to the tactic Correctness, this program generates 22 proof obligations. These obligations and their proofs are not listed here, for obvious space considerations, but they can be respectively regenerated and replayed using the source files mentioned above. We now detail the four subproblems of the formal proof, namely those of proving the correctness, the termination, the preservation and the fact that all subscripts are within the bounds.

3.1. Correctness

The formal proof of correctness strictly follows Hoare's one, and we keep the notation of [2] as much as we can. We first introduce the size N of the array and the subscript f, as two parameters of type Z, with the following axiom:

$$1 \le f \le N.$$

[1] This development is freely available on the web page of Coq users's contributions, at http://coq.inria.fr/contribs.

```
1.  let m = ref 1 in let n = ref N in
2.  while !m < !n do
3.   { invariant (m_invariant m A) ∧ (n_invariant n A)
4.         ∧ (permut A A@0) ∧ 1 ≤ m ∧ n ≤ N
5.     variant n − m }
6.   let r = A[f] in let i = ref !m in let j = ref !n in
7.   begin
8.    while !i ≤ !j do
9.     { invariant (i_invariant m n i r A) ∧ (j_invariant m n j r A)
10.        ∧ (m_invariant m A) ∧ (n_invariant n A) ∧ 0 ≤ j
11.        ∧ i ≤ N + 1 ∧ (termination i j m n r A) ∧ (permut A A@0)
12.      variant N + 2 + j − i }
13.     label L;
14.     while A[!i] < !r do
15.      { invariant (i_invariant m n i r A) ∧ i@L ≤ i ∧ i ≤ n
16.            ∧ (termination i j m n r A)
17.        variant N + 1 − i }
18.       i := !i + 1
19.     done;
20.     while !r < A[!j] do
21.      { invariant (j_invariant m n j r A) ∧ j ≤ j@L ∧ m ≤ j
22.            ∧ (termination i j m n r A)
23.        variant j }
24.       j := !j − 1
25.     done;
26.     assert { A[j] ≤ r ≤ A[i] };
27.     if !i ≤ !j then begin
28.      let w = A[!i] in begin A[!i] := A[!j]; A[!j] := w end;
29.      assert { (exchange A A@L i j) ∧ A[i] ≤ r ≤ A[j] };
30.      i := !i + 1;
31.      j := !j − 1
32.     end
33.    done;
34.    assert { m < i ∧ j < n };
35.    if f ≤ !j then
36.     n := !j
37.    else if !i ≤ f then
38.     m := !i
39.    else
40.     begin n := f; m := f end
41.  end
42.  done
43.  { (Found A) ∧ (permut A A@) }
```

Fig. 2. The final code and annotations of *Find*.

Notice that it implies in particular $1 \leq N$, which is implicitly used at several places in the original proof, and which will be explicitly used in the formal proof. Then we introduce the array A, as a global array of type (array N Z).

The postcondition of the program is introduced as a predicate *Found* on the array A, defined as follows:

$$(\mathit{Found}\ A) \quad \stackrel{\text{def}}{=} \quad \forall p, q : Z.\ 1 \leq p \leq f \leq q \leq N \Rightarrow A[p] \leq A[f] \leq A[q].$$

Then the program is given the postcondition (*Found* A), line 43. Remember that A denotes the *current* value of A, so its final value here.

The invariants over m and n (lines 3 and 4 in Fig. 1) are introduced by the following two definitions:

$$(m\_invariant\ m\ A) \stackrel{\text{def}}{=} m \le f \ \wedge\ \forall p, q : Z.\ 1 \le p < m \le q \le N \Rightarrow A[p] \le A[q]$$

$$(n\_invariant\ n\ A) \stackrel{\text{def}}{=} f \le n \ \wedge\ \forall p, q : Z.\ 1 \le p \le n < q \le N \Rightarrow A[p] \le A[q].$$

Those properties are inserted as invariants of the main loop (line 3). But they are also inserted as invariants of the middle loop (line 10). Indeed, the middle loop modifies the array A, and therefore the fact that the above two properties are also invariants of the middle loop must be explicitly expressed. At this point, one might notice that Hoare's keyword comment was rather informal: it clearly introduces a loop invariant but also the stronger property that it holds at some particular places inside the loop body, those places not being clearly stated. For instance, the property stated in line 3 of Fig. 1 is indeed also an invariant of the middle loop (line 11), because the middle loop only modifies the middle section, but this is not an obvious property. By using real loop invariants, whose meanings are clear, we suppress the ambiguity of the keyword comment.

Similarly, we introduce definitions for the invariants over i and j (lines 8 and 9 in Fig. 1):

$$(i\_invariant\ m\ n\ i\ r\ A) \stackrel{\text{def}}{=} m \le i \ \wedge\ \forall p : Z.\ 1 \le p < i \Rightarrow A[p] \le r$$

$$(j\_invariant\ m\ n\ j\ r\ A) \stackrel{\text{def}}{=} j \le n \ \wedge\ \forall q : Z.\ j < q \le N \Rightarrow r \le A[q].$$

One might wonder why $i\_invariant$ is a predicate over n and $j\_invariant$ a predicate over m: it will become clear in the next section, where these predicates will be extended in order to establish termination. The above properties about i and j are added to the invariant of the middle loop (line 9). The predicate $i\_invariant$ is also added as an invariant on the loop which increases i (line 15), and similarly the predicate $j\_invariant$ is added as an invariant of the loop which decreases j (line 21).

Each loop leads to two proof obligations: the first states that the invariant holds at the loop entrance, and the second states the preservation of the invariance property together with the decrease of the variant. For instance, the initialization of the main loop's invariant generates the obligation

$$(m\_invariant\ 1\ A) \ \wedge\ (n\_invariant\ N\ A)$$

which exactly combines Hoare's Lemmas 1 and 2, and its preservation produces three other obligations (due to the three cases at the end of the loop body) which correspond to Hoare's Lemma 6. Similarly, the initialization of the middle loop gives one obligation, corresponding to Lemmas 4 and 5, and the preservation of its invariant is expressed by two other obligations, corresponding to Lemmas 10 up to 13 in Hoare's paper. The correctness of the two inner loops on i and j are expressed by four obligations, corresponding to Lemmas 8 and 9. The establishment of the postcondition is expressed by one obligation, which is exactly Lemma 3.

The two assertions in the original code (lines 14 and 17 in Fig. 1) are kept in our program (lines 26 and 29). Although they are not absolutely needed, they help our understanding of the proof obligations, and allow us to keep close to the original proof. They give rise to four obligations, which are easily discharged.

3.2. Termination

In Hoare's paper, the problem of termination is considered in a separate paragraph. New invariants are added and five new lemmas are stated and proved. However, parts of the termination proof are done in a rather informal manner, without clear invariants. In our case, the termination of each loop is justified by a non-negative integer variant (lines 5, 12, 17 and 23), which strictly decreases at each execution of the loop body.

For the two inner loops, we follow Hoare's argument, showing that i and j are necessarily bound. This requires additional invariants for these variables, which express the existence of such bounds. For the i-loop, the predicate $i\_invariant$ is modified as follows:

$$(i\_invariant\ m\ n\ i\ r\ A) \stackrel{\text{def}}{=} \ldots \ \wedge\ (i \le n \Rightarrow \exists p : Z.\ i \le p \le n \ \wedge\ r \le A[p]).$$

There is a slight difference here from Hoare's argument: he uses the invariant $\exists p.\ i \leq p \leq n \ \wedge\ r \leq A[p]$ and shows that it is preserved in the middle loop, *provided that* $i \leq j$ *holds at the end of the loop body*. So it is not a real invariant, which would hold at the end of the loop, but only a property that holds *inside* the loop. Adding the guard $i \leq n$ in front of the property makes it a real invariant. The corresponding obligations are proved following Hoare's paper (Lemmas 14, 15 and 17). The definition of *j_invariant* is extended in a similar way:

$$(j\_invariant\ m\ n\ j\ r\ A) \quad \stackrel{\text{def}}{=} \quad \ldots \ \wedge\ (m \leq j \Rightarrow \exists q : Z.\ m \leq q \leq j \ \wedge\ A[q] \leq r).$$

The proof of termination of the middle loop is immediate, since either i is increased or j is decreased at each step. Therefore, the quantity $j - i$ always decreases and $N + 2 + j - i$ can be taken as variant. The corresponding proof obligations are rather trivial.

The proof of termination of the main loop is surely the most complex one. Indeed, although the variant $n - m$ is quite simple, the fact that it decreases strictly is quite difficult to establish. Hoare's very subtle argument relies on the fact that both $m < i$ and $j < n$ hold at the end of the middle loop (assertion line 34). Therefore, since either n is assigned the value of j or m is assigned the value of i, the distance between n and m decreases. To establish the assertion $m < i \ \wedge\ j < n$ at the end of the middle loop, Hoare shows that the conditional line 27 (if $!i \leq !j \ \ldots$) is always executed *at least once*. Indeed, the first time we encounter this conditional, $A[f]$ is still equal to r, and therefore the property $i \leq f \leq j$ holds. Although this is a perfectly correct argument, we cannot use it directly: There is no way to express in the specification that some execution of a loop body is the first one. The same property must be expressed by an invariant. This is achieved by introducing the following predicate *termination*:

$$(termination\ i\ j\ m\ n\ r\ A) \quad \stackrel{\text{def}}{=} \quad (i > m \ \wedge\ j < n) \ \vee\ (i \leq f \leq j \ \wedge\ A[f] = r)$$

which expresses that, either both i and j have been respectively increased and decreased, or they are still respectively on left and right sides of f, with $A[f]$ equal to r. This predicate is added in the invariants of the three inner loops (lines 11, 16 and 22) and we prove that it is preserved. Then we can prove the assertion $m < i \wedge j < n$ at the end of the middle loop (line 34), which is immediate since $(termination\ i\ j\ m\ n\ r\ A)$ and $i < j$ both hold.

3.3. Preservation

Preservation of the initial elements of the array is quickly treated at the end of Hoare's paper. His argument is simple: since the program only does exchanges of pairs of elements, and since any composition of exchanges is a permutation, it is obvious that we get a permutation of the initial array at the end of execution. Hoare seems satisfied with this informal argument, and even explains that a fully formalized one would be difficult to obtain and imply tedious proof obligations.

Actually, it depends on the formal definition of a permutation, and if the intuitive definition is used, this part of the proof can be very easy. Indeed, we can follow Hoare's argument, using the fact that

(1) the only modifications of A are exchanges of two elements;
(2) the reflexive, symmetric and transitive closure of exchanges is exactly the set of permutations.

We first define a predicate *exchange* which expresses that two arrays of N elements only differ by swapping the two elements at subscripts i and j:

$$\begin{array}{l}(exchange\ t\ t'\ i\ j) \quad \stackrel{\text{def}}{=} \\ \quad 1 \leq i, j \leq N \ \wedge\ t[i] = t'[j] \ \wedge\ t[j] = t'[i] \ \wedge \\ \quad \forall k : Z.\ (1 \leq k \leq N \ \wedge\ k \neq i \ \wedge\ k \neq j) \Rightarrow t[k] = t'[k].\end{array}$$

Notice that we impose for the subscripts i and j to be within the bounds of the arrays. Then we define a predicate *permut* which expresses that two arrays are permutations of each other. We define it *inductively*, as the smallest equivalence relation containing all the exchanges:

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

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

With this definition, the proof obligations are straightforward to prove and indeed follows the intuition quite closely: when the array A is not modified, the reflexivity of *permut* is used; when two modifications of the array A are done successively, the transitivity of *permut* is used; and when the array A is modified at line 28, two of its elements are swapped, making the property *exchange* true, and hence the property *permut*.

Finally, the predicate $(permut\ A\ A_{@})$ is added in the postcondition of the program, and in the invariants when necessary (lines 4 and 11). Notice the use of the notation $A_{@0}$ to refer to the initial value of A. We also add the assertion $(exchange\ A\ A_{@L}\ i\ j)$ in line 29 for convenience. All the related obligations are easily proved.

3.4. Correct accesses in arrays

This last problem, which consists in proving that all the indices used in the program are within the bounds 1 to N, was not treated by Hoare, although he noted the need of doing it. In our case, the corresponding proof obligations are automatically generated by the tactic Correctness, each time the array is accessed or modified. In order to be able to establish those proof obligations, we need additional invariants about m, n, i and j. First, we add the properties $1 \leq m$ and $n \leq N$ in the invariant of the main loop (line 4). Then we add the properties $0 \leq j$ and $i \leq N + 1$ in the invariant of the middle loop (line 10). And finally we express in the two inner loops that i stays within its initial value and n (line 15), and that j stays within m and its initial value (line 21). Notice the use of the label L to denote the initial values of i and j inside the middle loop. The proof obligations related to the indices are easily discharged by the arithmetical tactic of Coq, Omega, all the necessary inequalities being now available from the context.

4. Discussion

The main purpose of this paper was to demonstrate the adequacy of the system Coq for specifying and certifying imperative programs. We illustrated this ability with the proof of a non-trivial program, *Find*, following a handmade proof by Hoare [2]. The specification strictly followed the original one. In particular, Coq's notion of inductive predicates allowed us to define the permutation of two arrays as the smallest equivalence relation containing transpositions, and then to apply a simple argument from [2]. Even Hoare's tricky reasoning to establish the termination was easily translated into an invariant property. We finally ended up writing a 43 line program, including 17 lines of annotations.

When this annotated program was fed to Coq's total correctness tactic, 22 proof obligations were generated, and they included all Hoare's original 18 proof obligations. Following Hoare's proofs was then easy, although it required over 600 interactions with the system. The automatic decision procedure for linear arithmetic Omega [12] coming with the system Coq was found particularly useful (invoked 130 times in total). The overall development time is not very meaningful, since specification and proofs were given in Hoare's paper, but proofs of similar algorithms (Quicksort, Heapsort, Knuth–Morris–Pratt) have been realized in 2 or 3 days [5,8].

Obviously, such a formal proof can be conducted in any tool supporting basic imperative programming features. Regarding the discharge of proof obligations, only a decision procedure for Presburger arithmetic is really mandatory, which is now part of most proof assistants. To our knowledge, the formal proof of *Find* has only been done, beside Coq, with the Atelier B by Donzeau-Gouge [13]. The only real difficulty encountered was in the definition of the permutation predicate, somewhat cumbersome in the set-theoretical specification language of the B method. (This predicate also appear in various sorting examples from the VDM Examples Repository [14], and leads to quite tedious proof obligations.) One great advantage of using the Calculus of Inductive Constructions as a logic is the ability to define inductive predicates, such as the permutation predicate in this case. Conversely, the proof in the Atelier B was conducted by successful refinements, following closely Hoare's top-down construction. A similar refinement mechanism would be a great improvement to the tactic Correctness; it is work in progress.

Although Hoare succeeded in doing the proof of correctness of a quite complex program without any tool support – nor any mistake, which is a real achievement – and claimed that "it is hardly more laborious than the traditional practice of testing", he noticed himself in the conclusion of his paper that

> "In the future, it may be possible to enlist the aid of a computer in formulating the lemmas, and perhaps even in checking the proofs".

This has now become true, and Hoare's *Find* program appears as a good challenge for any method aiming at proving the correctness of imperative programs, since it is a rather complex program of small size, and the mathematical proofs are not too easy, so that it also tests the proof support.

Acknowledgements

This article was written while the author was an International Fellow at Computer Science Laboratory, SRI International (Menlo Park, CA), which provided a high-quality working environment. The author thanks the anonymous referees for the many suggested improvements to this paper. The author is also grateful to Christine Paulin for her help in finding the right invariant involved in the proof of totality.

References

[1] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580, 583, also in [15] pages 45–58.
[2] C.A.R. Hoare, Proof of a program: *Find*, Communications of the ACM 14 (1) (1971) 39–45, also in [15] pages 59–74.
[3] C.A.R. Hoare, Algorithm 65: Find, Communications of the ACM 4 (7) (1961) 321–322.
[4] J.-C. Filliâtre, Proof of imperative programs in type theory, in: International Workshop, TYPES '98, Kloster Irsee, Germany, in: Lecture Notes in Computer Science, vol. 1657, Springer-Verlag, 1998. URL: http://www.lri.fr/~filliatr/ftp/publis/types98.ps.gz.
[5] J.-C. Filliâtre, Preuve de programmes impératifs en théorie des types, Thèse de doctorat, Université Paris-Sud, July 1999. URL: http://www.lri.fr/~filliatr/ftp/publis/these.ps.gz.
[6] J.-C. Filliâtre, Verification of non-functional programs using interpretations in type theory, Journal of Functional Programming 13 (4) (2003) 709–745. English translation of [5]. URL: http://www.lri.fr/~filliatr/ftp/publis/jphd.ps.gz.
[7] Coq, The Coq Proof Assistant, 2001. http://coq.inria.fr/.
[8] J.-C. Filliâtre, N. Magaud, Certification of sorting algorithms in the system Coq, in: Theorem Proving in Higher Order Logics: Emerging Trends, 1999. URL: http://www.lri.fr/~filliatr/ftp/publis/Filliatre-Magaud.ps.gz.
[9] T. Coquand, G. Huet, The calculus of constructions, Information and Computation 76 (2–3) (1988) 95–120.
[10] C. Paulin-Mohring, Extracting F_ω's programs from proofs in the calculus of constructions, in: Sixteenth Annual ACM Symposium on Principles of Programming Languages, ACM, Austin, 1989.
[11] E. Giménez, Two approaches to the verification of concurrent programs in Coq, 1999, personal communication. URL: http://pauillac.inria.fr/~gimenez/papers.html.
[12] W. Pugh, The omega test: A fast and practical integer programming algorithm for dependence analysis, Communications of the ACM 35 (8) (1992) 102–114.
[13] V. Donzeau-Gouge, Proof of the Find algorithm with the B method, 1999. personal communication.
[14] The VDM examples repository. URL: http://www.csr.ncl.ac.uk/vdm/examples.
[15] C.A.R. Hoare, C.B. Jones, Essays in Computing Science, Prentice Hall, 1989.