Proving Correctness of Imperative Programs in Higher Order Logic *

Alfio Martini

[1]Av. Marechal Andrea 11/210
91340-400 – Porto Alegre – RS – Brazil

alfio.martini@gmail.com

Abstract. *Hoare Logic has a long tradition in formal verification and has been continuously developed and used to verify a broad class of programs, including sequential, object-oriented and concurrent programs. The purpose of this work is to provide a detailed and accessible exposition of the several ways that one can conduct, explore and write proofs of correctness of sequential imperative programs with Hoare logic and the Isabelle proof assistant. With the proof language Isar, it is possible to write structured, readable proofs that are suitable for human understanding and communication.*

1. Introduction

Using Hoare logic [Hoare 1969], we can prove that a program is correct by applying a finite set of inference rules to an initial program specification of the form $\{P\}\ c\ \{Q\}$, such that P and Q are logical assertions, and c is a imperative program or program fragment. The intuition behind such a specification, widely known as Hoare triple or as partial correctness assertion (PCA), is that if the program c starts executing in a state where the assertion P is true, then if c terminates, it does so in a state where the assertion Q holds. This program logic has a long tradition in formal verification and has been continuously developed and used to verify a broad class of programs, including sequential, object-oriented and concurrent programs [Apt et al. 2009].

On the other hand, modern proof assistants are mature tools with which several important mathematical problems are proved correct, and which are also being used as a support for the development of program logics libraries that can be used to certify software developments. Many interactive proof assistants like Isabele/HOL [Nipkow et al. 2002], Coq [Appel et al. 2014], Lean [de Moura et al. 2015] and verification tools like Why3 [Filliâtre 2013], just to name a few examples, provide ways to specify and prove programs using Hoare Logic with a great degree of automation.

In Isabelle/HOL, the program semantics and the proof systems are embedded into higher-order logic and then suitable tactics are formalized to reduce the amount of human interaction in the application of the proof rules. As far as possible, decision procedures are invoked to check automatically logical implications needed in the premises of the proof rules. A dedicated library provides great support for automation, a concrete syntax for the specification of Hoare triples, a verification condition generator, and a rich set of proof tactics and tools. Most importantly,

*Work completed: see at https://github.com/alfiomartini/hoare-imp-isab.

Isabelle provides a formal proof language called *Isar*, that supports highly readable, structured and detailed proofs in natural deduction style. Modern research, work and advertisement of the benefits of state of the art proof assistants tend to give a great emphasis on automation of the proof process, or at least parts of it. Even when automation works, a high level proof may be wanted, either because it is required for communication, certification, or for the simple joy of enlightenment. Thus the skill of proof construction, hopefully in language as natural as possible, is a craft that must be learned.

The purpose of this work is to provide a detailed and accessible exposition of the several ways that one can conduct, explore and write proofs of correctness of sequential imperative programs with Hoare logic and the Isabelle proof assistant. Besides that, we highlight a proof methodology base on proof scripts and high level structured proofs in Isar. The first is very helpful in proof exploration, while the second is fundamental to control proof complexity and to convey clear reasoning. As an example of this approach, we develop in detail a correctness proof of a non-trivial case study: the insertion sort algorithm.

The paper is organized as follows: in section 2 we introduce the basic concepts for proving correctness of programs in Isabelle with Hoare Logic. In section 3 we discuss a more substantial case study, insertion sort, and discuss the need to prove several auxiliary lemmas to achieve full automation. Finally, in section 4 we summarize the main ideas and contributions of this work. The complete report related to this work together with the corresponding Isabelle theories can be found at https://github.com/alfiomartini/hoare-imp-isab.

2. Hoare Logic in Isabelle/HOL

The purpose of this section is to introduce the basic concepts of Isabelle/HOL needed to read the paper and discuss the fundamental ideas underlining the formalization of Hoare Logic in `Isabelle/HOL`, i.e., the library formalized in the theory `HOL-Hoare`[1]. Isabelle is a generic meta-logical framework for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which stands for *Higher Order Logic*. HOL can be understood by the equation HOL = Functional Programming + Logic. Thus, most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight the essential notation. The space of total functions is denoted by the infix ⇒. Other type constructors, e.g., list, set, are written postfix, i.e., follow their argument as in `'a list`, where `'a` is a type variable.

The HOL-Hoare theory is an implementation of Hoare logic for a simple imperative language with commands for assignment, sequence, conditional choice and while loops. Each while loop must be annotated with an invariant. Hoare triples can be stated like goals of the form `VARS x y ... {P} prog {Q}`, where `prog` is a program in the language, `P` is the precondition, `Q` the postcondition. These assertions can be any formula in HOL, which are written in standard logical syntax. The prefix `x y ...` is the list of all program variables in `prog`. The latter list must be nonempty

[1] https://isabelle.in.tum.de/dist/library/HOL/HOL-Hoare/

and it must include all variables that occur on the left-hand side of an assignment in `prog`.

The implementation hides reasoning in Hoare logic completely and provides a method (`vcg`) for transforming a goal in Hoare logic into an equivalent list of verification conditions. Verification conditions are purely logical formulas, not containing program constructs. They can be checked or discharged using any standard proof tool (theorem prover or proof assistant) with support for the data types of the language. The function that maps an annotated Hoare triple to its set of verifications conditions is easily defined by structural induction in HOL (e.g., see [Gordon 1988]). If it is desired to simplify the resulting verification conditions at the same time, the user can apply the method `vcg_simp`. This implementation is a logic of partial correctness. You can only prove that your program does the right thing if it terminates, but not that it terminates.

```
lemma imp_pot:
"VARS (a::int) (b::nat) (p::int) (i::nat)
{a=A ∧ b=B}
i := 0; p := 1;
WHILE i<b
  INV { p = a^i ∧ i≤ b ∧ a=A ∧ b = B}
  DO p := p * a;i:=i+1 OD
{p = A^B}"
apply (vcg)
apply (auto)
done
```

Figure 1. Power Algorithm in Isabelle

A Hoare triple for a program that computes the positive power of an integer is formalized in `HOL-Hoare` as shown in Figure 1. Isabelle automatically computes the type of each variable in a term. Despite type inference, it is sometimes necessary to attach explicit **type constraints** to a variable or term. The syntax is $\mathtt{t} :: \tau$ as in $\mathtt{n} :: \mathtt{nat}$. In our example, we have added type constraints just to improve readability. The sequence of `apply` commands right after the end of the triple is called a *proof script*. It is a procedural low level language that is very useful to explore initial proof attempts, especially when the proof depends on a number of additional lemmas. Proof scripts is a document model for unstructured proofs and can only be understood if you are playing the script inside Isabelle.

After applying the tactic `vcg`, the proof state in Figure 2 shows the three verification conditions that must be proved. Applying the automatic proof method `auto` solves the three goals. Essentially, auto tries to simplify the subgoals. If it fails, it leaves to the user simplified versions of the most difficult cases.

In Isabelle, we have the universal quantifier $\bigwedge$ and the implication $\Longrightarrow$. They are part of the Isabelle framework, not of the logic HOL. They are used essentially to express generality (the state notion of an arbitrary value), inference rules, and convey structure to proof states, i.e., as means to separate assumptions from con-

```
proof (prove)
goal (3 subgoals):
 1. ⋀a b p i. a = A ∧ b = B ⟹ 1 = a ^ 0 ∧ 0 ≤ b ∧ a = A ∧ b = B
 2. ⋀a b p i.
       (p = a ^ i ∧ i ≤ b ∧ a = A ∧ b = B) ∧ i < b ⟹
       p * a = a ^ (i + 1) ∧ i + 1 ≤ b ∧ a = A ∧ b = B
 3. ⋀a b p i.
       (p = a ^ i ∧ i ≤ b ∧ a = A ∧ b = B) ∧ ¬ i < b ⟹ p = A ^ B
```

Figure 2. Power Algorithm in Isabelle - VC's

```
definition is_perm::"'a list ⇒ 'a list ⇒ bool"
where "is_perm l1 l2 ≡ length l1 = length l2
                      ∧ (∀x. count x l1 = count x l2)"

lemma inss_hoare: "VARS xs ys :: ('a::linorder) list
 {xs=X}
 ys:=[];
 WHILE xs ≠ []
    INV {isorted ys ∧ is_perm X (ys @ xs)}
 DO ys := ins (hd xs) ys; xs := tl xs OD
{isorted ys ∧ is_perm X ys}"
```

Figure 3. Hoare Triple - Insertion Sort

clusions. The prefix $\bigwedge\ a\ b\ p\ i$ means "for arbitrary a, b, p, i (of appropriate types)". Right-arrows of all kinds always associate to the right. An iterated implication like $\mathtt{A_1} \Longrightarrow \mathtt{A_2} \Longrightarrow \ldots \Longrightarrow \mathtt{A_n} \Longrightarrow \mathtt{B}$ will indicate a proof judgment $A_1, \ldots, A_n \vdash B$ with assumptions $A_1, \ldots, A_n$ and conclusion B.

3. Case Study: Insertion Sort

In this section we discuss the validity of a partial correctness assertion for the insertion sort algorithm. We first present the functional version of insertion sort and discuss some important properties of this algorithm before we proceed with the proof of the triple itself. Lists in HOL are of type `'a list` and are built up from the empty list [ ] via the infix constructor # for adding an element at the front. In the case of non-empty lists, functions `hd` and `tl` return the first element and the rest of the list, respectively. Two lists are appended with the infix operator @.

The Hoare triple for insertion sort is shown in Figure 3. It states that for every arbitrary input list X, if the program terminates, then the output list ys is sorted and it is also a permutation of the input list X. X is a logical, ghost variable used to record the input value for the program variable xs. We consider two lists a permutation of one another if they have the same length and the same number of occurrences of elements for each list element.

In Figure 4 we show the basic functions for our development. The function

`ins` inserts an element at the right position in a ordered list. Note the restriction `'a :: linorder` on the type variable `'a`. The polymorphism is restricted to the types which are instances of the *type class* `linorder`, i.e., only to those types which can provide an ordering predicate that satisfy the axioms of a total order, i.e., a partial order in which every pair of elements can be compared. The introduction of type classes in Isabelle was strongly influenced by the analogous concept in the programming language `Haskell`. The funcion `iSort` returns a sorted list by repeated application of the function `ins`. It could be defined directly by `foldr ins [ ]`. The function `le` receives an element and a list and returns `True` if and only if the element is the least element in the list. The number of occurrences of an element in a list is computed by `count`. A list is sorted if and only if every element is the least when compared to all its successors.

```
fun ins::"'a::linorder ⇒ 'a list ⇒ 'a list" where
  "ins x [] = [x]" |
  "ins x (y # ys) =
      (if x ≤ y then (x # y # ys) else y#ins x ys)"

fun iSort::"('a::linorder) list ⇒ 'a list" where
"iSort [] = []" |
"iSort (x # xs) = ins x (iSort xs)"

fun le::"('a::linorder) ⇒ 'a list ⇒ bool"  where
"le x [] = True" |
"le x (y # ys) = (x ≤y ∧ le x ys)"

fun isorted::"('a::linorder) list ⇒ bool" where
"isorted [] = True" |
"isorted (x # xs) = (le x xs ∧ isorted xs)"

fun count:: "'a ⇒ 'a list ⇒ int" where
"count x [] = 0" |
"count x (y # ys) =(if x=y then  1 + count x ys else count x ys)"
```

Figure 4. Functions for Insertion Sort

To help Isabelle in the proof of this triple we need a small set of properties related to the functions for insertion sort defined earlier. They are show in Figure 5 and are proved by induction. The informal meaning of these lemmas can be understood as follows:`le_ins` states if a certain value precedes all elements of a list and also precedes another value `a`, the it also precedes all the elements of the list which includes `a`. Lemma `le_mon` says that the function $\lambda w.\ le\ w\ xs$ is monotonic w.r.t. to the order relation. Proposition `ins_sorted` asserts that the insert function preserves sortedness, while `is_sorted` claims that insertion sort always returns a sorted list. Lemma `ins_count` asserts counting is compatible with insertion of new elements, and `count_sum` proves that counting the number of occurrences is compatible with concatenation of lists. Proposition `len_sort` says that the length of an input list is invariant under insertion sort, while `count_isort` affirm that the number of occurrences of elements is invariant under sorting. Finally, `ins_len` states that the lenght of list is compatible with insertion of new elements.

```
lemma le_ins: "le x (ins a xs) = (x ≤ a ∧ le x xs)"
lemma le_mon:"x≤y ⟹ le y xs ⟹ le x xs"
lemma ins_sorted: "isorted (ins a xs) = isorted xs"
lemma is_sorted:"isorted(iSort xs)"
lemma ins_count:
 "count x (ins k xs) = (if x = k then 1 + count x xs else count x xs)"
lemma count_sum:"count x (xs @ ys) = count x xs + count x ys"
lemma len_sort:"length(iSort xs) = length xs"
lemma count_iSort: "count x (iSort xs) = count x xs"
lemma ins_len:"length (ins k xs) = 1 + length xs"
```

Figure 5. Insertion Sort Lemmas

Isabelle automatic tactic can be very handy in helping the user discover what are the missing lemmas that must be proved, since `auto` solves the easy stuff and leaves the harder ones for the user to figure out. After calling the verification condition generator we are left with the following proof goals:

```
proof (prove)
goal (3 subgoals):
 1. ⋀(xs::'a list) ys::'a list.
       xs = X ⟹ isorted [] ∧ is_perm X ([] @ xs)
 2. ⋀(xs::'a list) ys::'a list.
       (isorted ys ∧ is_perm X (ys @ xs)) ∧ xs ≠ [] ⟹
       isorted (ins (hd xs) ys) ∧ is_perm X (ins (hd xs) ys @ tl xs)
 3. ⋀(xs::'a list) ys::'a list.
       (isorted ys ∧ is_perm X (ys @ xs)) ∧ ¬ xs ≠ [] ⟹
       isorted ys ∧ is_perm X ys
variables:
  X :: 'a list
```

The first and third verification condition are easily seen to be true. The first because every empty list is sorted by definition, the empty list [ ] is neutral with respect to concatenation, and by equality elimination, every list is a permutation of itself. The third because the assumption $\neg xs \neq [\,]$ is equivalent to $xs = [\,]$. Then the conclusion follows by equality elimination and for the fact that the empty list [ ] is neutral with respect to concatenation. The second verification condition seems the real challenge, specially for the second conjunct. A proof script that solves the three goals is shown bellow.

```
apply (vcg)
apply (auto simp add:is_perm_def) — ‹ 1 ›
   apply (simp add: ins_sorted)  — ‹ 2 ›
   apply (simp add: ins_len) — ‹ 3 ›
   apply (smt count.simps(2) count_sum hd_Cons_tl ins_count) — ‹ 4 ›
done
```

Because two of the three verification conditions are trivial, we try to prove everything automatically with `apply (auto simp add:is_perm_def)`. However, `auto`

does not solve the second subgoal and generates three new subgoals. These new subgoals are solved by the three (indented) subsequent invocations of `apply`. Note the inclusion of lemmas in the simplifier set. However, proof scripts cannot be understood unless you are playing the proof yourself in `Isabelle`. In this case, in particular, the last proof command uses a call to SMT solvers (e.g., CVC4 and Z3) to solve the remaining subgoal. The outcome of the `smt` command depends on tools external to Isabelle, so it can be hard to predict if they will prove the same things in the future or if they will even still be available in an Isabelle-compatible form in a number of years.

In fact, this proof was discovered by *Sledgehammer*, an Isabelle's subsystem for harnessing the power of first-order automatic theorem provers. Given a conjecture, it heuristically selects a few hundred relevant facts (lemmas, definitions, or axioms) from Isabelle's libraries, translates them to first-order logic along with the conjecture, and delegates the proof search to external resolution provers (E, SPASS and Vampire) and SMT solvers (CVC3, and Z3).

Unless we attempt a structured proof with Isar, it can be difficult to remove these `smt` calls. Trial and error is never a good advice. Even if we come up with another proof script that avoids the aforementioned issues, still we would not know the explanation, the reasoning that justifies why the partial correctness assertion is valid. This may be desired or even essential if we want to communicate our reasoning so that users and readers can properly appreciate and understand the logical entailments underlying a particular program or algorithm.

```
proof (vcg)
   fix xs ys
   assume ass:"(isorted ys ∧ is_perm X (ys @ xs)) ∧ xs ≠ []"
   show "isorted (ins (hd xs) ys)
         ∧ is_perm X ((ins (hd xs) ys) @ tl xs)"
    proof (rule conjI)
       show "isorted (ins (hd xs) ys)" sorry
    next
       have pg1:"length X  =  length ((ins (hd xs) ys) @ tl xs)"
            sorry
       have pg2:"∀ k. count k X = count k (ins (hd xs) ys @ tl xs)"
            sorry
       from pg1 pg2 show "is_perm X (ins (hd xs) ys @ tl xs)" sorry
   qed
qed (auto simp add:is_perm_def)
```

Figure 6. Insertion Sort - Proof Draft

So we proceed now with a structure proof of the second verification condition, i.e., with the proof that the invariant is maintained by the loop. A proof sketch for this goal is shown in Figure 6. The proof of the whole verification condition is enclosed in the outermost `proof`...`qed` delimiters. The argument for the `proof` command is the verification condition generator. The next three lines show the typical structure `fix`...`assume`...`show`... of Isar proofs: `fix` is used to declare arbitrary variables, `assume` to state the assumptions and `show` to assert the goal of the proof. The proof argument `rule conjI` applies the natural deduction rule for conjunction introduction to the proof goal stated in the outermost `show` command. The

command `sorry` means `by cheating`. This method solves its goal without actually proving it and indicate that this step must later be refined with a real proof. It is a fake proof pretending to solve the pending claim without further ado. However, if used wisely, can be very helpful for top-down development of structured proofs. The auxiliary propositions labeled `pg1,pg2` state the two necessary conditions to prove that the loop maintains the property that the two lists are a permutation of one another. The command `auto simp add:is_perm_def` after the outermost `qed` solves automatically the remaining goals, i.e., the first and third verification conditions.

In Figure 7 we in detail the first part of the proof, in very small reasoning steps. The whole proof itself is somehow long, but it carries detailed explanations of why the algorithm works.

```
proof (rule conjI)
   from ass have "isorted ys" by simp
   from this show "isorted (ins (hd xs) ys)" by (simp add:ins_sorted)
next
  from ass have 1:"is_perm X (ys @ xs)" and 2:"xs ≠ []" by auto
  from 2 have hdtl:"xs = hd xs # tl xs" by simp
  from 1 have 3:"∀ x. count x X = count x (ys @ xs)" by (simp add:is_perm_def)
  have pg1:"length X  =  length ((ins (hd xs) ys) @ tl xs)"
     proof -
         from 1 have 4:"length X = length (xs @ ys)" by (simp add:is_perm_def)
         also have "...= length xs + length ys" by simp
         also have "... =  1 + length ys + length xs - 1" by simp
         also have "... = length (ins (hd xs) ys) + length (tl xs)"
                          by (simp add: "2" ins_len)
         also have "... = length ((ins (hd xs) ys) @ tl xs)" by simp
         finally show ?thesis  by simp
     qed
```

Figure 7. Insertion Sort - Structured Proof

This proof uses a special Isar construct to write proofs in the *calculation style*, i.e, when the steps are a chain of equations or inequations. The three dots is the name of an unknown that Isar automatically instantiates with the right-hand side of the previous equation. There is an Isar theorem variable called `calculation`, similar to `this` (remember that `this` variable holds the proposition proved in the previous step). When the first `also` in a chain is encountered, Isabelle sets `calculation := this`. In each subsequent `also` step, Isabelle composes the theorems `calculation` and `this` (i.e. the two previous equations) using transitivity of equality. The command `finally` is a shorthand for `also from calculation`. Thus, in the `finally` step, the chain of equations is closed with respect to the transitivity rule and it is stored in the variable `calculation`. The unknown `?thesis` is implicitly matched against the most recent assertion stated `show` command.

4. Concluding Remarks

In this report I have tried to convey a detailed and accessible presentation to several techniques available for reasoning with Hoare Logic in the proof assistant Isabelle. The implementation of Hoare logic that comes with the standard distribution of the system provide a standard syntax to declare Hoare triples and a varied set of proof

tactics for proof automation. Moreover, the user can write programs over a rich collection of data types which are formalized in higher order logic.

Despite Isabelle sophisticated tools for proof automation (especially through Sledgehammer), being able to construct structured, human-readable reasoning about program code is fundamental to communicate ideas properly. It is also a craft that is important to learn and develop, not only to tackle more complex cases but also to understand properly the subtle logical behavior of more sophisticated algorithms and programs. The use of the proof language `Isar` for documentation, reasoning, and especially communication at a high-level, is essential.

References

Appel, A. W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., and Leroy, X. (2014). *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA.

Apt, K. R., de Boer, F., and Olderog, E.-R. (2009). *Verification of Sequential and Concurrent Programs*. Springer Publishing Company, Incorporated, 3rd edition.

de Moura, L. M., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. (2015). The lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388.

Filliâtre, J.-C. (2013). One Logic To Use Them All. In Bonacina, M. P., editor, *CADE 24 - the 24th International Conference on Automated Deduction*, Lake Placid, NY, United States. Springer.

Gordon, M. J. C. (1988). *Programming language theory and its implementation - applicative and imperative paradigms*. Prentice Hall International series in Computer Science. Prentice Hall.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg. Available at http://isabelle.in.tum.de/documentation.html.