Reasoning about Imperative Programs in Higher Order Logic

Alfio Martini

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 the user 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.

Keywords. Hoare logic, Interactive Theorem Proving, Higher Order Logic, Formal Methods Teaching.

Contents

1. Introduction

Program verification is a systematic approach to proving the correctness of programs. Correctness means that the programs enjoy certain desirable properties. For sequential programs these properties are delivery of correct results and termination. For concurrent programs, those with several active components, the properties of interference freedom, deadlock freedom and fair behavior are also important.

Using Floyd/Hoare logic [16, 11], one 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 [3, 1]

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. Therefore, it is meaningful to ask whether program verification cannot be carried out automatically. Unfortunately, the theory of computability tells us that fully automatic verification of program properties is in general an undecidable problem. However, for Hoare logic, much of the proof process can be automated through the process of computing verification conditions. Many interactive proof assistants like Isabele/HOL [15], Coq [2], Lean [6] and verification tools like Why3 [7], 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, Isabelle provides a formal proof language called *Isar*, that supports 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 report 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 based on proof scripts and high level structured proofs in Isar. The first are 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 non-trivial case study: the insertion sort algorithm.

This text is organized as follows: in section 2 we provide a concise, yet formal presentation of Hoare Logic, with special attention to semantical concepts. Section 3 presents the underlying ideas about automation of Hoare Logic and in section 4 we introduce the basic concepts for proving correctness of programs in Isabelle with Hoare Logic. In section 5 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 6 we summarize the main ideas and contributions of this work. This document together with the corresponding Isabelle theories can be found at `https://github.com/alfiomartini/hoare-imp-isab`.

2. Background

The material in this section is for the most part, well-known, and it is included here in order to fix notation and to improve readability.

2.1. Hoare Logic: Syntax and Semantics

The central feature of Hoare logic are the *Hoare triples* or, as they are often called, *partial correctness assertions*. We use both expressions interchangeably. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form $\{P\}\ c\ \{Q\}$, where P, Q can be assertions in a specification language or predicates written in standard mathematical language, and $c \in \boldsymbol{Prg}$ is a program fragment in a imperative language. P is called the precondition and Q the postcondition of the triple.

The imperative language we consider has the usual constructors for assignment, sequential composition, conditional command and the identity program. We assume a countably infinite set $\boldsymbol{Var}$ of program variables, ranged over by metavariables $x, y, \ldots$. We consider programs that use a finite number of program variables and assume there are enough of them available for any program. The abstract syntax of the syntactic category $\boldsymbol{Prg}$ is given by the following productions, where c_0, c_1, c range over $\boldsymbol{Prg}$, a over arithmetic expressions, and b range over boolean expressions. The precise syntax of arithmetic and boolean expressions is standard and can be found elsewhere [17].

$$c \in \boldsymbol{Prg} \quad ::= \quad \underline{\mathbf{skip}} \mid x := a \mid c_0 ; c_1 \mid \underline{\mathbf{if}}\ b\ \underline{\mathbf{then}}\ c_0\ \underline{\mathbf{else}}\ c_1\ \underline{\mathbf{fi}} \mid \underline{\mathbf{while}}\ b\ \underline{\mathbf{do}}\ c\ \underline{\mathbf{od}}$$

An informal understanding about the meaning of a Hoare triple can be given as follows: *If P holds in the initial state, and if the execution of c terminates when started in that state, then Q will hold in the state in which c halts.* Note that for $\{P\}\ c\ \{Q\}$ to hold, we do not require that S halts when started in states satisfying P, but that if it does halt, then Q holds in the final state. As an example, taken from [10], we have the following partial correctness assertion to compute the power of A^B of an integer A and non-negative integer B.

$$\begin{array}{l}\{a = A\ \wedge\ b = B \wedge B \geq 0\}\\ i := 0; p := 1; \underline{\textbf{while}}\ i < b\ \underline{\textbf{do}}\ p := p * a; i := i + 1\ \underline{\textbf{od}}\\ \{p = A^B\}\end{array} \tag{2.1}$$

The lower case variables are the *state* or *program variables.* The uppercase variables are the so-called *logical, ghost* or integer variables. Logical variables are often used as parameters, to remember the initial values of program variables.

In this section we assume that both logical and program variables range over the integers, but in Isabelle/HOL they may range over the rich collection of types provided by HOL and polymorphic types as well (see section 4). Thus, a notion of satisfaction for a Hoare triple has to take into account values for both logical and program variables. For the first case we use *environments* and for the second, *states.* An environment for the (logical) integer variables is a function $env : \boldsymbol{IVar} \to \mathbb{Z}$, where $\boldsymbol{IVar}$ is a countably infinite set of integer logical variales. The set of all such enviroments is $Env = (\boldsymbol{IVar} \to \mathbb{Z})$.

In order to evaluate an expression or to define the execution of a command, we need the notion of a *memory state.* A memory state σ is an element of the set Σ, which contains all functions from program variables to integers: $\sigma \in \Sigma = (\boldsymbol{Var} \to \mathbb{Z})$. Given a state σ, we denote by $\sigma[x \mapsto n]$ the memory where the value of x is updated to n, i.e.,

$$\sigma[x \mapsto n](y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{if } y \neq x \end{cases}$$

We assume a basic satisfaction relation between states, environments and formulas in the specification logic. The judgment $\sigma \models_{env} P$ states that P holds in the state $\sigma : Loc \to \mathbb{Z}$ with respect to the environment $env : \boldsymbol{IVar} \to \mathbb{Z}$.

A program assertion P is valid in an environment $env : \boldsymbol{IVar} \to \mathbb{Z}$, written $\models_{env} P$, iff $\forall \sigma \in \Sigma.\ \sigma \models_{env} P$. A program assertion P is called (arithmetic) *valid*, written $\models P$, iff $\forall\ env : \boldsymbol{IVar} \to \mathbb{Z}.\ \models_{env} P$. We say that Q is a logical consequence of P in an environment $env : \boldsymbol{IVar} \to \mathbb{Z}$, written $P \models_{env} Q$, iff $\forall \sigma \in \Sigma.\ \sigma \models_{env} P \to \sigma \models_{env} Q$. Moreover, we say that Q is a logical consequence of P, written $P \models Q$ iff $\forall \sigma \in \Sigma.\ \forall\ env : \boldsymbol{IVar} \to \mathbb{Z}.\ \sigma \models_{env} P \to \sigma \models_{env} Q$.

In the following we assume that the semantics of programs is given by an inductive evaluation relation $\Downarrow_p \subseteq \Sigma \times \boldsymbol{Prg} \times \Sigma$. By an expression

$\langle c, \sigma \rangle \Downarrow_p \sigma'$ we mean that the execution of program c from the initial state σ leads to the final state σ' (see e.g, [17, 13]). We say that a triple $\{P\}\ c\ \{Q\}$ is true at a state $\sigma \in \Sigma$ and environment $env : \textbf{\textit{IVar}} \to \mathbb{Z}$, written $\sigma \models_{env} \{P\}\ c\ \{Q\}$ iff $\forall \sigma' \in \Sigma.\ \sigma \models_{env} P \to (\langle \sigma, c \rangle \Downarrow_p \sigma' \to \sigma' \models_{env} Q$. The triple is valid in an environment $env : \textbf{\textit{IVar}} \to \mathbb{Z}$, written $\models_{env} \{P\}\ c\ \{Q\}$ iff $\forall \sigma \in \Sigma.\ \sigma \models_{env} \{P\}\ c\ \{Q\}$. Finally, a partial correctness assertion is (arithmetic) *valid*, written $\models \{P\}\ c\ \{Q\}$, iff $\forall\ env : \textbf{\textit{IVar}} \to \mathbb{Z}.\ \models_{env} \{P\}\ c\ \{Q\}$. Note that arithmetic validity is a formula in *higher order logic*, since we quantity (universally) both over environments and states.

2.2. Hoare Logic: Proof Calculus

The following rules of the *Hoare Proof Calculus* define inductively the ternary relation $\vdash$. This relation is defined by triples, where the first and third components are assertions in a given specification logic (first or higher order logic) and the second component is an imperative program. These triples can be seen as theorems of the proof calculus. The expression $Q[x/a]$ means the simultaneous replacement of every occurrence of the program variable x in the assertion Q by the arithmetic expression a.

$$\frac{}{\vdash \{P\}\ \underline{\textbf{skip}}\ \{P\}}\ \textsf{Skip} \qquad\qquad \frac{}{\vdash \{Q[x/a]\}\ x := a\ \{Q\}}\ \textsf{Ass}$$

$$\frac{\vdash \{P\}\ c_1\ \{Q\} \quad \vdash \{Q\}\ c_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}\ \textsf{Comp}$$

$$\frac{\vdash \{P \wedge B\}\ c_1\ \{Q\} \quad \vdash \{P \wedge \neg B\}\ c_2\ \{Q\}}{\vdash \{P\}\ \underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ c_0\ \ \underline{\textbf{else}}\ c_1; \underline{\textbf{fi}}\ \{Q\}}\ \textsf{IfE} \qquad \frac{\vdash \{P \wedge B\} c \{P\}}{\vdash \{P\}\ \underline{\textbf{while}}\ b\ \underline{\textbf{do}}\ c\ \underline{\textbf{od}}\ \{P \wedge \neg B\}}\ \textsf{PWh}$$

$$\frac{\vdash P \to Q \quad \vdash \{Q\}\ c\ \{R\}}{\vdash \{P\}\ C\ \{R\}}\ \textsf{Stren} \qquad\qquad \frac{\vdash \{P\} C \{Q\} \quad \vdash Q \to R}{\vdash \{P\}\ C\ \{R\}}\ \textsf{Weakn}$$

The power of Floyd/Hoare treatment of imperative programs [11, 16] lies in its use of variable substitution to capture the semantics of assignment: $P[x/a]$, the result of replacing every free occurrence of program variable x in P by expression a, is the precondition which guarantees that assignment $x := a$ will terminate in a state satisfying P. At a stroke, difficult semantic questions that have to do with stores and states are converted into simpler syntactic questions about first-order logical formula. The rule Ass can be understood as follows: if initially $P[x/a]$ is true, then after the assignment x will have the value of a, and hence no substitution is necessary anymore, i.e., P itself is true afterwards. The rule PWh deserves a more detailed explanation. The truth of a triple $\{P\}\ c\ \{Q\}$ depends on the state and in general, we do not know how many times (if ever) the loop body will execute for each given initial state, and thus we cannot predict the final state after the loop finishes. It will change after each execution of the body. Therefore, we cannot specify the functional behaviour of a loop with arbitrary assertions P and Q. In the rule, the assertion P denotes an invariant assertion, i.e., a relation between the program variables that remain constant during loop execution. The rules says that if executing c once preserves the truth of P, then executing c any number of times also preserves the truth of P. Thus *a loop invariant* is a property of a program loop that is true before (and after) each iteration and it can be considered a proper specification of the behaviour of the loop construct. The consequence rules of precondition strengthening (Stren) and postcondition weakening (Weakn) are the rules that connect the proof system of the underlying specification language with the Hoare calculus itself. The formal presentation of the remaining rules match our intuitive understanding of the programming constructs.

Let $\{P\}\ c\ \{Q\}$ be a partial correctness assertion. Then the Hoare calculus is sound, i.e., every theorem is a valid formula.

$$\vdash \{P\}\ c\ \{Q\} \quad \text{only if} \quad \models \{P\}\ c\ \{Q\}$$

Example 2.1. *Using the proof rules, we can organize the proof of example 2.1 according to the following proof tree, where*

$$\begin{array}{ll}
w \triangleq \underline{\textbf{\textit{while}}}\ i < b\ \underline{\textbf{\textit{do}}}\ body\ \underline{\textbf{\textit{od}}} & \\
init \triangleq i := 0; p := 1 & body \triangleq p := p * a; i := i + 1; \\
Pre \triangleq a = A \land b = B \land B \geq 0 & Pos \triangleq p = A^B \\
bw \triangleq i < b & INV \triangleq p = a^i \land i \leq b \land a = A \land b = B
\end{array}$$

$$\dfrac{\dfrac{\vdots\ \boxed{1}}{\vdash \{Pre\}\ init\ \{INV\}} \qquad \dfrac{\dfrac{\dfrac{\vdots\ \boxed{2}}{\vdash \{INV \land bw\}\ body\ \{INV\}}}{\vdash \{INV\}\ w\ \{INV \land \neg bw\}} \qquad \vdots\ \boxed{3}}{\vdash \{INV\}\ w\ \{Pos\}}}{\vdash \{Pre\}\ init; w\ \{Pos\}}$$

where $\boxed{3}$ *corresponds to the proof tree of the judgment* $\vdash INV \land \neg bw \to Pos$.

□

The above proof tree tell us that to prove the original triple, it is sufficient to solve the three proof obligations indicated by the vertical dots.

Using the rules of assignment and then the rule composition, we can reduce the first two proof obligations to a set of verification conditions in the specification logic. For instance, we can give the following linear presentation for $\boxed{2}$, where $AB \triangleq a = A \land b = B$ and $IAB \triangleq i + 1 \leq b \land AB$:

1	$\{p = a^{i+1} \land i + 1 \leq b \land AB\}\ i := 1 + 1\ \{INV\}$	Ass
2	$\{p * a = a^{i+1} \land IAB\}\ p := p * a\ \{p = a^{i+1} \land IAB\}$	Ass
3	$\{p * a = a^{i+1} \land IAB\}\ p := p * a; i := i + 1\ \{INV\}$	Comp(1, 2)
4	$INV \land bw$	
5	$\vdots$	
6	$p * a = a^{i+1} \land IAB$	

The implication outlined in the proof box above could be proved, for instance, like this:

1	$INV \wedge bw$	Prem
2	$i < b$	$\wedge$ E(1)
3	$i + 1 \leq b$	Theorem(2)
4	$p = a^i =$	$\wedge$E(1)
5	$p * a = a^{i+1}$	Theorem(4)
6	$p * a = a^{i+1} \wedge i + 1 \leq b$	$\wedge$E(5, 3)
7	$INV \wedge bw \rightarrow p * a = a^{i+1} \wedge i + 1 \leq b$	$\rightarrow$ I(1 − 6)

whete Theorem denote basic laws of arithmetic. Thus, to prove the original triple, it is sufficient to prove the following verification conditions:

$$\begin{array}{l} 1. \vdash\ a = A \wedge b = B \wedge B \geq 0 \rightarrow a^0 = 1 \wedge b \geq 0 \wedge a = A \wedge b = B \\ 2. \vdash INV \wedge i < b \rightarrow p * a = a^{i+1} \wedge i + 1 \leq b \wedge a = A \wedge b = B \\ 3. \vdash INV \wedge \neg bw \rightarrow Pos \end{array} \tag{2.2}$$

These three proof obligations correspond to the following claims about the loop invariant:

1. The invariant should be true initially;
2. INV should be an invariant;
3. The invariant should be strong enough to imply the postcondition.

These assertions are arithmetic valid and also provable in any reasonable presentation of the theory of integer arithmetic [5, 12].

3. On Automation of Hoare Logic

In the following section we discuss annotated specifications, verification conditions and how this automation process is formalized in Isabelle/HOL.

3.1. General Idea and Derived Rules

From the small example shown in 2.1 we can clearly see that proofs are typically long and boring. Besides, there are a lot of complicated details to get right (formal proofs of the verification conditions). Also, in practice, we work with the proof system in a backwards way: starting from the goal of show $\{P\}\ c\ \{Q\}$, one generates subgoals, subsubgoals, etc., until the problem is solved.

A typical system for Hoare Logic for automation of Hoare Logic takes as input a partial correctness specification (Hoare Triple) annotated with logical assertions describing relationships between variables. From the annotated specification, the system generates a set of purely mathematical statements, called *verification conditions* (VC's). The verification conditions are passed to a *theorem prover* program, which attempts to prove them automatically. If it fails, advice is asked from the user.

Although proof rules presented in section rules are sufficient for all proofs, the rules for *assignment, skip command* and *while loop* are inconvenient: they can only be applied backwards if the pre- or postcondition are of a special form. Thus, in searching for a technique for automation of Hoare Logic, the following derived rules are preferred, since they can be applied backwards without regard to the form of the pre- and postconditions.

$$\frac{\vdash P \rightarrow Q}{\vdash \{P\}\ \underline{\mathbf{skip}}\ \{Q\}}\ \mathsf{Skip'} \qquad \frac{\vdash P \rightarrow Q[x/a]}{\vdash \{P\}\ x := a\ \{Q\}}\ \mathsf{Ass'}$$

$$\frac{\vdash \{P \wedge b\}\ c\ \{P\} \quad \vdash P \wedge \neg b \rightarrow Q}{\vdash \{P\}\ \underline{\mathbf{while}}\ b\ \underline{\mathbf{do}}\ c\ \underline{\mathbf{od}}\ \{Q\}}\ \mathsf{PWh'}$$

These derived rules are the ones that are formalized in the Isabelle library `Hoare-Logic` (see section 4).

3.2. Annotated Specifications and Verification Conditions

Annotated commands are the central idea behind the development of automated tools for establishing the validity of partial correctness assertions. Thus, the syntactic set of annotated command is defined by the following grammar:

$$\begin{aligned} c \ ::= \ & \underline{\mathbf{skip}} \mid x := a \mid c_0; x := a \mid c_0; \{D\}c_1 \mid \\ & \underline{\mathbf{if}}\ b\ \underline{\mathbf{then}}\ c_0\ \ \underline{\mathbf{else}}\ c_1; \underline{\mathbf{fi}} \mid \underline{\mathbf{while}}\ b\ \underline{\mathbf{do}}\ \{D\}c \end{aligned}$$

In set of productions above, x is a program variable, a an arithmetic expression, b is a boolean expression, c, c_0, c_1 are annotated commands and D is an assertion such that in $c_0; \{D\}c_1$, c_1 is not an assignment. Note that in a sequence of commands $c_0; c_1$, it is unnecessary to do this when c_1 is an assignment $x := a$, because in this case an annotation can be derived from the postcondition. In an annotated while loop $\underline{\mathbf{while}}$ b $\underline{\mathbf{do}}$ $\{D\}c$, the assertion D is intended to be an invariant. An *annotated partial correctness* has the form $\{P\}$ c $\{Q\}$, where c is an annotated command. Ignoring the annotations, an annotated command is an ordinary command. An annotated partial correctness assertion is *valid* when its associated unannotated Hoare triple is.

Example 3.1. *The annotated partial correctness specification corresponding to example 2.1 is shown bellow, where $INV \triangleq p = a^i \wedge i \leq b$.*

$$\begin{array}{l} \mathit{powerAnn} \triangleq \\ \{a = A\ \wedge\ b = B \wedge B \geq 0\} \\ i := 0; p := 1; \\ \{INV\} \\ \underline{\mathbf{\mathit{while}}}\ i < b\ \underline{\mathbf{do}}\ \{INV\}\ p := p * a; i := i + 1\ \underline{\mathbf{\mathit{od}}} \\ \{p = A^B\} \end{array} \tag{3.1}$$

Note that we are entitled to use the invariant as an annotation before the loop, since the invariant is always true at the start of the while construct.

□

That not every annotated assertion is valid, is clear. In order to be so, it is sufficient to establish the validity of certain assertions, called *verification conditions*, where all mention of commands is removed. 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 defined by structural induction on annotated commands as follows:

$$\begin{array}{lcl}
vc(\{P\}\ \underline{\mathbf{skip}}\ \{Q\}) & = & \{P \to Q\}\\
vc(\{P\}\ x := a\ \{Q\}) & = & \{P \to Q[x/a]\}\\
vc(\{P\}\ c_0; x := a\ \{Q\}) & = & vc(\{P\}\ c_0\ \{Q[x/a]\})\\
vc(\{P\}\ c_0; \{D\}\ c_1\ \{Q\}) & = & vc(\{P\}\ c_0\ \{D\}) \cup vc(\{D\}\ c_1\ \{Q\})\\
 & (c_1 \text{ not an assignment}) & \\
vc(\{P\}\ \underline{\mathbf{if}}\ b\ \underline{\mathbf{then}}\ c_0\ \ \underline{\mathbf{else}}\ c_1; \underline{\mathbf{fi}}\ \{Q\}) & = & vc(\{P \wedge b\}\ c_0\ \{Q\})\\
 & & \cup\, vc(\{P \wedge \neg b\}\ c_1\ \{Q\})\\
vc(\{P\}\ \underline{\mathbf{while}}\ b\ \underline{\mathbf{do}}\ \{D\}c\ \{Q\}) & = & vc(\{D \wedge b\}\ c\ \{D\})\\
 & & \ \cup \{P \to D\} \cup \{D \wedge \neg b \to Q\}
\end{array}$$

Using this definition on the example 3.1, we have:

$$\begin{array}{l}
vc(powerAnn)\\
= vc(\{Pre\}\ i := 0; p := 1\ \{INV\}) \cup vc(\{INV\}\ w\ \{p = A^B\})\\
= \{Pre \to 1 = a^0 \wedge 0 \le b \wedge AB\} \cup \{INV \to INV\}\\
\quad \cup\, vc(\{INV \wedge bw\}\ body\ \{INV\}) \cup \{INV \wedge \neg bw \to p = A^B\}\\
= \{Pre \to 1 = a^0 \wedge 0 \le b \wedge AB\}\\
\quad \{INV \wedge bw \to p * a = a^{i+1} \wedge i + 1 \le b \wedge AB\}\\
\quad \{INV \wedge \neg bw \to p = A^B\}
\end{array}$$

where

$$\begin{array}{ll}
w \triangleq \underline{\mathbf{while}}\ i < b\ \underline{\mathbf{do}}\ body\ \underline{\mathbf{od}} & \\
AB \triangleq a = A \wedge b = B \wedge B \ge 0 & body \triangleq p := p * a; i := i + 1;\\
Pre \triangleq a = A \wedge b = B \wedge B \ge 0 & Pos \triangleq p = A^B\\
bw \triangleq i < b & INV \triangleq p = a^i \wedge i \le b \wedge a = A \wedge b = B
\end{array}$$

It can be shown that for an arbitrary annotated partial correctness assertion $\{P\}\ c\ \{Q\}$ to be valid, it is sufficient that its verification conditions are valid (see [8]).

4. 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 to present the essential ideas of Hoare Logic in `Isabelle/HOL` as it is formalized in the library `HOL-Hoare`[1].

Using Floyd/Hoare logic [16, 11], 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. 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.

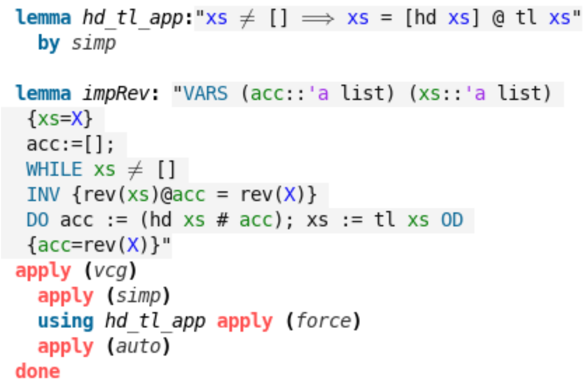
Isabelle is a generic meta-logical framework for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which stands for *Higher Order Logic* [14]. HOL can be understood by the equation $\mathsf{HOL} = \mathsf{Functional\ Programming} + \mathsf{Logic}$. Thus, most 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 $\Rightarrow$. Other type constructors, e.g., list, set, are written postfix, i.e., follow their argument as in `'a set`, where `'a` is a type variable. 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 @. Function `rev` reverses a list. In HOL, types and terms must be enclosed in double quotes.

The HOL-Hoare theory is an implementation of Hoare logic for a simple imperative language with assignments, null command, conditional, sequence 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 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 in HOL. The 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.

Figure 1 presents a Hoare triple for a program that computes the reversal of a list.

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

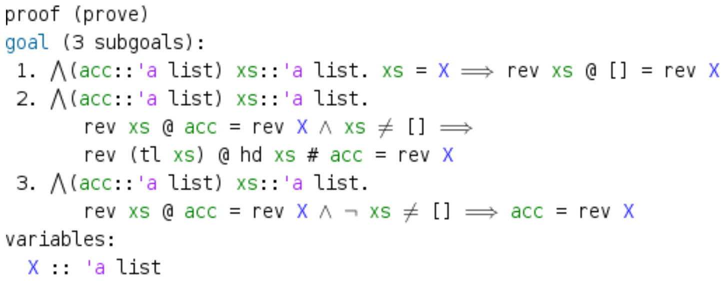


```
lemma hd_tl_app:"xs ≠ [] ⟹ xs = [hd xs] @ tl xs"
  by simp

lemma impRev: "VARS (acc::'a list) (xs::'a list)
  {xs=X}
  acc:=[];
  WHILE xs ≠ []
  INV {rev(xs)@acc = rev(X)}
  DO acc := (hd xs # acc); xs := tl xs OD
  {acc=rev(X)}"
apply (vcg)
  apply (simp)
  using hd_tl_app apply (force)
  apply (auto)
done
```

Figure 1. Hoare Triple - List Reversal

In the precondition, the logical variable `X` is used to save the initial value of the program variable `xs`. In the loop, a new value of the accumulator is set with the head of the list is appended to its old value, while the new list is updated with its tail. The loop invariant asserts an essential relation between the program and logical variables during the iteration process: appending the reverse of the current value of the list with the value of the accumulator always gives the same result as the reverse of the initial input list. This invariant is strong enough to entail the postcondition, which asserts that the final value of the accumulator holds the reversal of the input list.

The sequence of `apply` commands after the postcondition comprises a proof script in Isabelle. This is a procedural language with which the user can issue commands, tactics and rule applications that work on the proof state. In this case, after the application of the verification condition generator, the proof state looks like this:

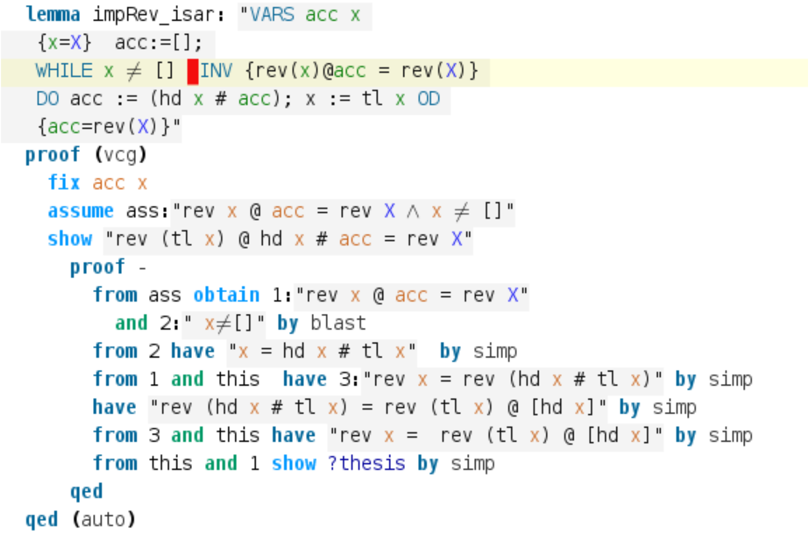


```
proof (prove)
goal (3 subgoals):
 1. ⋀(acc::'a list) xs::'a list. xs = X ⟹ rev xs @ [] = rev X
 2. ⋀(acc::'a list) xs::'a list.
       rev xs @ acc = rev X ∧ xs ≠ [] ⟹
       rev (tl xs) @ hd xs # acc = rev X
 3. ⋀(acc::'a list) xs::'a list.
       rev xs @ acc = rev X ∧ ¬ xs ≠ [] ⟹ acc = rev X
variables: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

In Isabelle's meta-logic, 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 a means to separate assumptions from conclusions. 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 $\mathtt{A}_1 \Longrightarrow \mathtt{A}_2 \Longrightarrow \ldots \Longrightarrow \mathtt{A}_\mathtt{n} \Longrightarrow \mathtt{B}$ will indicate a proof judgment $A_1, \ldots, A_n \vdash B$ with assumptions $A_1, \ldots, A_n$ and conclusion B.

The three verification conditions correspond to the following properties: the invariant is true after initialization (or before the loop starts), the invariant is maintained by the loop, and the invariant is strong enough to entail the postcondition. The three are discharged with automatic proof tactics. The first with the simplifier, the third with proof method `auto` (classical reasoning with simplification). The second is solved by another proof tool, `force` (classical reasoner with exhaustive proof search), which is fed with a lemma that states that every non-empty list can be factored as its head appended to its tail.

The first verification condition is true by equality elimination and the fact that the empty list is an identity with respect to concatenation. The third follows from the fact that the reverse of an empty list is empty and because the empty list is an identity with respect to concatenation. The third follows from the lemma `hd_tl_app` and the identity `rev` (`as@bs`) = `rev bs@rev as`. Using the isar proof language, we show a detailed proof of the second verification condition of the imperative list reversal in Figure 2.



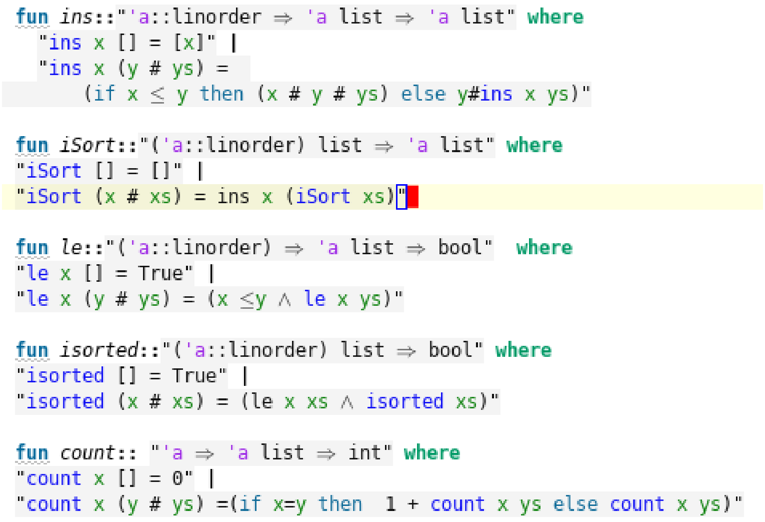
```
lemma impRev_isar: "VARS acc x
 {x=X}  acc:=[];
 WHILE x ≠ [] INV {rev(x)@acc = rev(X)}
 DO acc := (hd x # acc); x := tl x OD
 {acc=rev(X)}"
proof (vcg)
  fix acc x
  assume ass:"rev x @ acc = rev X ∧ x ≠ []"
  show "rev (tl x) @ hd x # acc = rev X"
    proof -
      from ass obtain 1:"rev x @ acc = rev X"
        and 2:" x≠[]" by blast
      from 2 have "x = hd x # tl x"  by simp
      from 1 and this  have 3:"rev x = rev (hd x # tl x)" by simp
      have "rev (hd x # tl x) = rev (tl x) @ [hd x]" by simp
      from 3 and this have "rev x =  rev (tl x) @ [hd x]" by simp
      from this and 1 show ?thesis by simp
    qed
qed (auto)
```

Figure 2. Structured Proof of Verification Condition

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 innermost proof. . . qed delimiters enclose the actual proof the last show statement. The argument − for proof indicates that we are not applying any proof method or rule to change the current proof state. The first line breaks the compound assumption into its two conjuncts. The second line uses the simplifier to prove that every non-empty list can be factored as its head followed by its tail. The predefined name this can be used to refer to the proposition proved in the previous step. The proof of the third step follows by congruence of function application. The fourth by the theorem rev(xs@ys) = (revys)@(revxs). The fifth step follow by transitivity of equality. In the last step, the unknown ?thesis is matched against the last declared show. The last show statement proves the actual goal and it also follows from transitivity of equality. The auto after the outermost qed proves automatically the remaining verification conditions (the first and third).

5. Case Study: Insertion Sort

In this section we discuss the validity of a partial correctness assertion for the insertion sort algorithm. We follow the same approach taken when discussing the imperative version of a list reversal. 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. In Figure 3 we show the basic functions for our development.

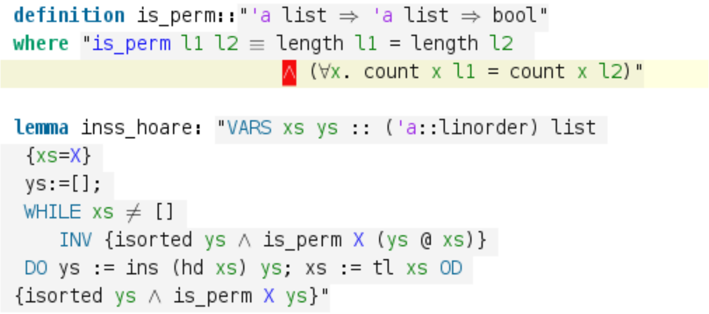


```
fun ins::"'a::linorder ⇒ 'a list ⇒ 'a list" where
  "ins x [] = [x]" |
  "ins x (y # ys) =
      (if x ≤ y then (x # y # ys) else y#ins x ys)"

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 3. Functions for Insertion Sort

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 [9]. 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.

The Hoare triple for insertion sort is shown in Figure 4. 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.

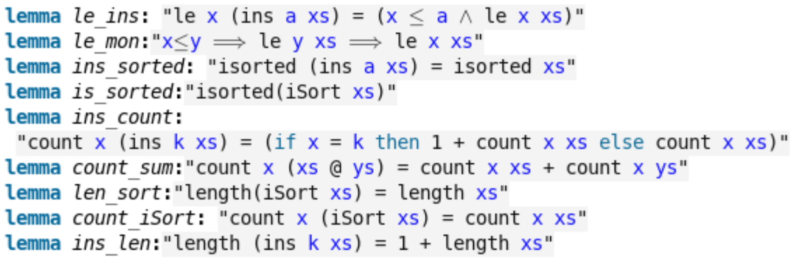


```
definition is_perm::"'a list ⇒ 'a list ⇒ bool"
where "is_perm l1 l2 ≡ length l1 = length l2
                      ∧ (∀x. count x l1 = count x l2)"

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 4. Hoare Triple - Insertion Sort

To help Isabelle in proving 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 propositions is outlined below.

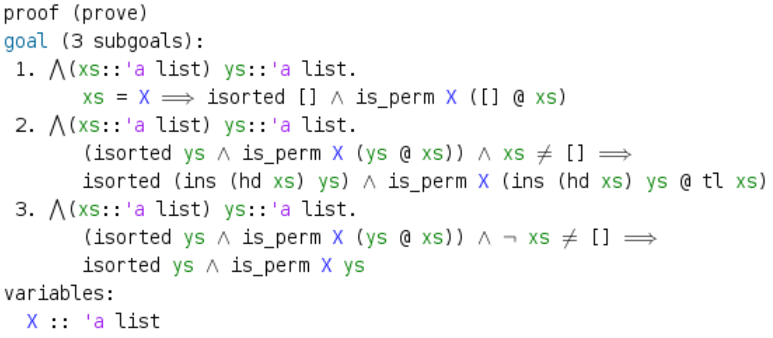


```
lemma le_ins: "le x (ins a xs) = (x ≤ a ∧ le x xs)"
lemma le_mon:"x≤y ⟹ le y xs ⟹ le x xs"
lemma ins_sorted: "isorted (ins a xs) = isorted xs"
lemma is_sorted:"isorted(iSort xs)"
lemma ins_count:
 "count x (ins k xs) = (if x = k then 1 + count x xs else count x xs)"
lemma count_sum:"count x (xs @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

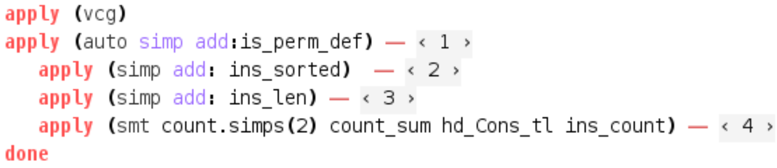
lemmas	Informal Meaning
le_ins	if a certain value precedes all elements of a list and also precedes another value a, the it also precedes all the elements of the list which includes a.
le_mon	the function $\lambda w.\ le\ w\ xs$ is monotonic w.r.t. to the order relation.
ins_sorted	the insert function preserves sortedness.
is_sorted	insertion sort always returns a sorted list.
ins_count	counting is compatible with insertion of new elements.
count_sum	counting the number of occurrences is compatible with concatenation of lists.
len_sort	the length of an input list is invariant under insertion sort.
count_isort	the number of occurrences of elements is invariant under sorting.
ins_len	the lenght of list is compatible with insertion of new elements.

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. A introductory exercise that discusses how we can discover the right lemmas can be found in [15], chapter 2. 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.

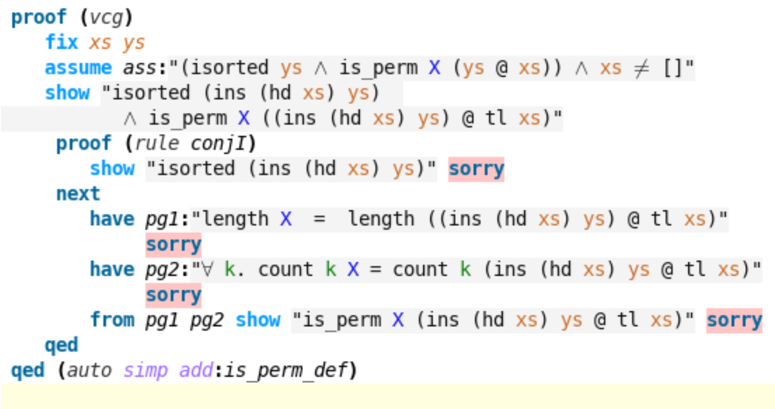


```
apply (vcg)
apply (auto simp add:is_perm_def) — ‹ 1 ›
   apply (simp add: ins_sorted)  — ‹ 2 ›
   apply (simp add: ins_len) — ‹ 3 ›
   apply (smt count.simps(2) count_sum hd_Cons_tl ins_count) — ‹ 4 ›
done
```

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. There are some important observations about this proof script:

1. Proof scripts cannot be understood unless you are playing the proof yourself in `Isabelle`.
2. The tactic `auto` works on all subgoals simultaneously and fails to solve them completely. As a proof draft, it is acceptable, but it should never be used like this in a final version. It's perfectly fine, however, when auto solves its goals completely, e.g. as terminal proof method, which it is not the case above.
3. The last proof command uses a call to SMT solvers (CVC4, 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* [4] and it can be difficult to remove these `smt` calls. Trial and error should be out of the question.
4. Even if we come up with another proof script that avoids the aforementioned issues, we would still not know the explanation, the reasoning that justifies why the partial correctness assertion is valid.

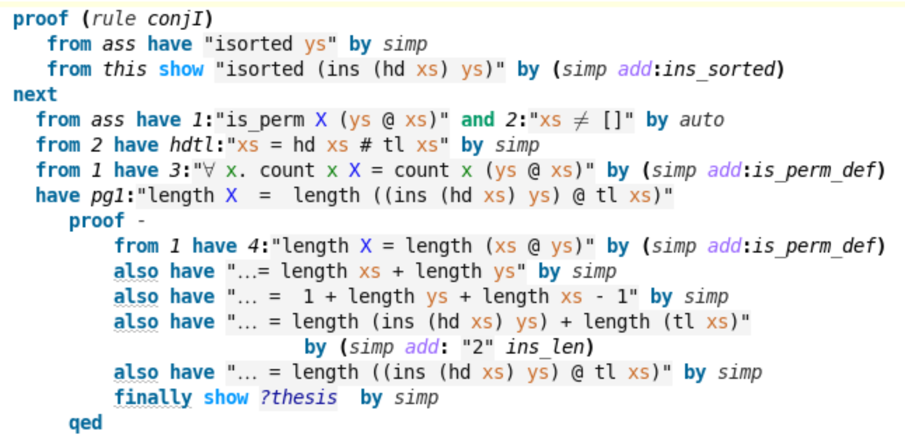
A high-level, human-readable proof may be desired or even essential if we want to communicate our reasoning so that users and readers can properly appreciate and understand the logical entailments underlying a particular program or algorithm. So we proceed now with a structure proof of the second verification condition, i.e., with the proof that the invariant is maintained by the loop. A proof sketch with the proof language Isar for this goal is shown in Figure 6. In the proof draft we see that the outermost structure is of a conjunction. The command `sorry` means `by cheating`. This method solves its goal without actually proving it and indicate that this step



```
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

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 command `rule conjI` applies the natural deduction rule for conjunction introduction to the proof goal stated in the outermost `show` command. 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.



```
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

In Figure 7 we formalize in Isar 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.

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 assertion stated by the previous `have` command.

6. Concluding Remarks

Programmers and software developers alike often see the subject of Hoare Logic as a tedious and impractical reasoning tool, despite being a key foundation of program verification. Thus it is essential to present them with modern tools that improve application as well as teaching of this essential concept for program and algorithm verification.

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

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[2] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers.* Cambridge University Press, New York, NY, USA, 2014.

[3] Krzysztof R. Apt, Frank de Boer, and Ernst-Rdiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer Publishing Company, Incorporated, 3rd edition, 2009.

[4] Jasmin Blanchete. User's Guide to Sledgehammer. http://isabelle.in.tum.de/documentation.html, June 2018.

[5] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag, Berlin, Heidelberg, 2007.

[6] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 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, 2015.

[7] Jean-Christophe Filliâtre. One Logic To Use Them All. In Maria Paola Bonacina, editor, *CADE 24 - the 24th International Conference on Automated Deduction*, Lake Placid, NY, United States, June 2013. Springer.

[8] Michael J. C. Gordon. *Programming language theory and its implementation - applicative and imperative paradigms.* Prentice Hall International series in Computer Science. Prentice Hall, 1988.

[9] Florian Haftmann. Haskell-style type classes with Isabelle/Isar. http://isabelle.in.tum.de/documentation.html, June 2019.

[10] J. Hein. *Discrete Structures, Logic, and Computability.* Jones & Bartlett Learning, 2010.

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

[12] Jacques Loeckx, Kurt Sieber, and Ryan D. Stansifer. *The Foundations of Program Verification.* John Wiley & Sons, Inc., New York, NY, USA, 1984.

[13] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications - a Formal Introduction.* Wiley professional computing. Wiley, 1992.

[14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL.* Springer Publishing Company, Incorporated, 2014.

[15] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002. Available at http://isabelle.in.tum.de/documentation.html.

[16] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, 19, 01 1967.

[17] Glynn Winskel. *The Formal Semantics of Programming Languages - an Introduction.* Foundation of computing series. MIT Press, 1993.

Alfio Martini
Porto Alegre - Brazil
e-mail: alfio.martini@gmail.com