Reasoning about Imperative Programs in Higher Order Logic

A. R. 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 presentation on how to use the Hoare Logic library that comes with the Isabelle distribution in order to conduct and write proofs of correctness of sequential imperative programs. Special attention is given to the structured proof language *Isar*, in which the user can write down highly readable proofs in a formal language that is amenable to human understanding, both for communication and maintenance.

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 [17, 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 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. Why not feed a program and its specification into a computer and wait for an answer? 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 [16], Coq [2], Lean [6] and verification tools like Why3 [7], just to name a few examples, provide ways to specify and prove programs using of Hoare Logic with a great degree of automation.

The purpose of this report is to provide a detailed and hopefully accessible presentation on how to use the Hoare Logic library that comes with the Isabelle distribution in order to conduct and write proofs of correctness of sequential imperative programs. 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.

The library provides great support for automation, a concrete syntax for the specification of Hoare triples, a verification condition generator, a rich set of proof tactics and tools. Most importantly, a structured proof language called *Isar*, that supports highly readable 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. However, apart from trivial cases, this is rarely possible. Even when automation works, the proof itself may be wanted, either because it is required for certification or for the simple joy of enlightenment. Thus the skill of construction proofs, hopefully in language as natural as possible, is a craft that must be learned, especially for students and the novices who want to try their chances with this essential technology.

The paper 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. Section 5 works in detail through an imperative version of list reversal and introduces the structured proof language *Isar*. In section 6 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 7 we summarize the main ideas developed in this work and list its main contributions.

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 [19].

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

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* or integer variables. Logical variables are used as parameters, to remember the initial values of program variables. They may appear on the right of an assignments expressions but never as the target variable in an assignment statement (left value). Therefore their values remain constant during program execution. Thus, both arithmetic and boolean expressions may include logical and program variables. If the logical variables never occur in the program, then they are also called *ghost 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).

As consequence, 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* 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 $\Downarrow_p \subseteq \Sigma \times \textbf{\textit{Prg}} \times \Sigma$. 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, [19, 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 relation $\vdash \subseteq \textbf{\textit{Assn}} \times \textbf{\textit{Prg}} \times \textbf{\textit{Assn}}$, i.e., the triples which can be see as theorem 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.

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

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

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

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

The power of Floyd/Hoare treatment of imperative programs [11, 17] 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 execution. 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*

$w \triangleq \textbf{\textit{while}}\ i < b\ \textbf{\textit{do}}\ body\ \textbf{\textit{od}}$
$init \triangleq i := 0; p := 1$ $\quad body \triangleq p := p * a; i := i + 1;$
$Pre \triangleq a = A \wedge b = B \wedge B \geq 0$ $\quad Pos \triangleq p = A^B$
$bw \triangleq i < b$ $\quad INV \triangleq p = a^i \wedge i \leq b \wedge a = A \wedge b = B$

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

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

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 $AB \triangleq a = A \wedge b = B$ and $IAB \triangleq i + 1 \leq b \wedge AB$:

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

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.

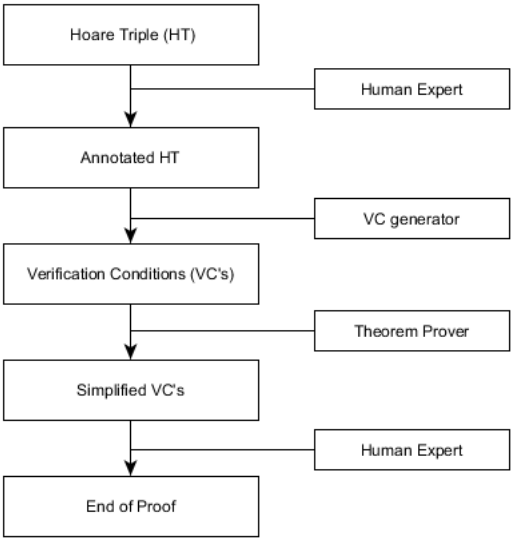

FIGURE 1. A Proof Checker for Hoare Logic

The diagram of Figure 1 shows the architecture of a theorem prover for Hoare Logic (from [8]). The system 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.

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

$$\frac{\vdash \{P\}\ c\ \{P\} \quad \vdash P \wedge \neg b \to Q}{\vdash \{P\}\ \underline{\textbf{while}}\ b\ \underline{\textbf{do}}\ c\ \underline{\textbf{od}}\ \{Q\}}\ \textsf{PWh'}$$

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:

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

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 $\underline{\textbf{while}}\ b\ \underline{\textbf{do}}\ \{D\} c$, 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* $INV \triangleq p = a^i \wedge i \leq b$.

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

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. This function is defined by structural induction on annotated commands as follows:

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

Using this definition on the example 3.1, we have:

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

where

$$
\begin{aligned}
&w \triangleq \textbf{while}\ i < b\ \textbf{do}\ body\ \textbf{od}\\
&AB \triangleq a = A \wedge b = B \wedge B \geq 0 \qquad body \triangleq p := p * a; i := i + 1;\\
&Pre \triangleq a = A \wedge b = B \wedge B \geq 0 \qquad Pos \triangleq p = A^B\\
&bw \triangleq i < b \qquad INV \triangleq p = a^i \wedge i \leq b \wedge a = A \wedge b = B
\end{aligned}
$$

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 a formalization of Hoare Logic in `Isabelle/HOL`, i.e., the library formalized in `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* [15]. 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 the simple imperative language introduced in section 2.1. 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

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

list of verification conditions in HOL. If it desired to simplify the resulting verification conditions at the same time, the user can apply the method `vcg_simp`. 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.

```
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 2. Power Algorithm in Isabelle

The example 2.1 that computes the positive power of an integer is formalized in `HOL-Hoare` as shown in Figure 2. 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 3 shows the three verification conditions that must be proved (compare 2.2). 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 the logic HOL. They are used essentially for generality (the state notion of an arbitrary value) and judgments or inference rules, respectively. Thus, 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.

```
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 3. Power Algorithm in Isabelle - VC's

5. A Preliminary Example: List Reversal

We consider now the case of writing an imperative algorithm for reversal of a list. The tail recursive function definition and two lemmas relating the tail recursive version and the (naive) recursive version `rev` ::′ `a list` ⇒′ `a list` are shown in Figure 4.

```
fun tail_rev::"'a list ⇒ 'a list ⇒ 'a list" where
"tail_rev [] acc = acc" |
"tail_rev (x#xs) acc = tail_rev xs (x # acc)"

lemma tail_rev:"∀ys. tail_rev xs ys = rev xs @ ys"
apply (induction xs)
apply (auto)
done

lemma "tail_rev xs [] = rev xs" by (simp add:tail_rev)
```

FIGURE 4. Reverse by Tail Recursion

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. It is defined in the library as follows:

$$\begin{aligned} &\texttt{rev} ::' \texttt{a list} \Rightarrow' \texttt{a list} \\ &\texttt{rev}\ [\,] = [\,] \\ &\texttt{rev}\ (\texttt{x\#xs}) = (\texttt{rev xs}) \mathbin{@} [\texttt{x}] \end{aligned} \tag{5.1}$$

In HOL, types and terms must be enclosed in double quotes. In the tail recursive version, the second parameter has the purpose of both holding an initial value and also to serve as an accumulator. The first parameter is the list to be reversed. In each call the first element of the list is left appended onto the second parameter.

The first lemma relates the two versions of list reversal and is easily proved by induction. The second lemma is a special case of the first and it states that the tail version reverses the first list if we initialize the second parameter with the empty list. The second lemma is proved with the proof method `simp`, which does term rewriting with the equational definitions of the functions. In the second lemma, we add the lemma `tail_rev` to the simplifer set of equations.

Base on the tail recursive reverse function, an imperative version is easily programmed with a while loop and it is shown in the Hoare triple of Figure 5.

```
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 5. Hoare Triple - List Reversal

After the application of the verification condition generator, the proof state looks like the one shown in Fig 6.

```
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
```

FIGURE 6. Hoare Triple - List Reversal VC's

The first verification condition is solved by the simplifier, and the third by using the automatic tactic `auto`. The second goal was harder and its proof was actually discovered by *Sledgehammer* [4]. In the proof discovered by Sledgehammer for the second verification condition, the lemma `hd_tl_app` is used. This lemma is proved in the beginning of Figure 5 and states that non-empty lists can be factored in its head and tail. The `force` method applies the classical reasoner and simplifier to the current goal. Unless it can prove the goal, it fails.

Sledgehammer is 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). Sledgehammer is very effective and has achieved great popularity with users in general.

Automation is essential for accomplishing complex proofs. However, using proof automatic tactics without having a clue about they solve a goal is far from ideal. Understanding should never be delegated.

So far we have used the script (procedural) language to write down our proofs. It is completely unstructured, i.e., it is comprised of a list of commands `apply (proof method)`) and can be understood if you play the proof inside Isabelle and watch what happens with the proof state at each step. However, one of the great advantages of Isabelle is that it supports a very rich language to write structured proofs in a style similar to natural deduction proofs ([14], chapter 4).

Isar stands for *Intelligible semi-automated reasoning* and was developed by Markus Wenzel in his PhD thesis [18]. The Isar proof language provides a general framework for human-readable natural deduction proofs. The two key features of Isar are: it is structured, not linear; it is readable without running it because you need to state what you are proving at any given point. Isar proofs are like structured programs with comments. A proof in Isar can be either compound (`proof-qed`) or atomic (`by`). A typical proof skeleton looks like this:

proof
  assume "ass"
  have "..." `by` (*method*)
  ⋮
  have "..." `by` (*method*)
  show "concl" `by` (*method*)
qed

This proves the conjecture "`ass` ⇒ `concl`″ . The intermediate `have's` are only there to bridge the gap between the assumption and the conclusion and do not contribute to the theorem being proved. On the other hand, `show` establishes the conclusion of the theorem. A *method* is a proof method and

it is an argument to the command `by`. Proof methods includes `rule` (which performs a backwards step with a given rule, unifying the conclusion of the rule with the current subgoal and replacing the subgoal by the premises of the rule), `simp` (for simplification), `auto` (for automation), `blast` (for predicate calculus reasoning), `force` (classical reasoner with simplification) and many others. Using the isar proof language, we show a detailed proof of the second verification condition of the imperative list reversal in Figure 7.

```
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 7. 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).

Also, Sledgehammer often fails when trying to discover proofs for complex assertions. In this case Isar is also very handy, because it allows us to break a complex proposition into smaller, understandable chunks, that can then be fed into Sledgehammer or other automatic proof methods of Isabelle.

6. 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 8 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)"

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

FIGURE 8. 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

```
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 9. Hoare Triple - Insertion Sort

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 final function provides a tail recursive version for the insertion sort algorithm.

The Hoare triple for insertion sort is shown in Figure 9. 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. 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 10 and are proved by induction.

```
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 10. Insertion Sort Lemmas

The informal meaning of these propositions is outlined below.

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 [16], 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 simultaneousl 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* and, unless we attempt a structured proof with Isar, its is difficult to know to remove these smt calls. Trial and error is out of the question.
4. 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.

```
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 11. 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 11. In the proof draft we see that the outermost structure is of a conjunction. The command `sorry` means `by cheating` and 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.

In Figure 12 we show a detailed proof of condition *pg*2, the first case. The whole proof itself is somehow long, but it carries detailed explanations of why the algorithm works.

```
have pg2:"∀ k. count k X = count k (ins (hd xs) ys @ tl xs)"
   proof (rule allI)
     fix k
     have "count k X = count k ((ins (hd xs) ys) @ tl xs)"
        proof (cases "k= hd xs")
           assume case1: "k = hd xs"
           have "count k ((ins (hd xs) ys) @ tl xs) =
                 count k (ins (hd xs) ys) + count k (tl xs)"
                    by (simp add:count_sum)
           also have "... = 1 + count k ys + count k (tl xs)"
               by (simp add:case1 ins_count)
           also have "... =  count k ys + 1 + count k (tl xs)" by simp
           also have "...= count k ys + count k ((hd xs) # tl xs)"
                by (simp add: case1)
           also have "...= count k ys + count k xs"  using hdtl by auto
           also have "...= count k (ys @ xs)" by (simp add:count_sum)
           also have "...= count k X" by (simp add: "3")
           finally show "count k X = count k ((ins (hd xs) ys) @ tl xs)"
              by simp
        next
```

FIGURE 12. 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`.

7. Concluding Remarks

Today's students and software developers alike 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 the `Isabelle/HOL` library `HOL-Hoare` for reasoning about imperative programs. Together, the library itself and the Isabelle/HOL 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. This work can be also be used as an introduction to theorem proving and formal methods within a specific domain of application.

Despite Isabelle sophisticated tools for proof automation (especially through Sledgehammer), we argued that a blind application of automation proof procedures without careful reasoning will do very little in improving education and training for students and developers alike. Therefore, we have emphasized the structured proof language `Isar` for proof documentation, reasoning, maintenance, and especially communication to readers and users alike. This is fundamental in the more interesting and hard cases. This rich set of tools for proof (and disproof) recommends Isabelle as an interactive theorem proving system for teaching, reasoning and programming as well.

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. Programming and Proving with Isabelle/HOL. http://isabelle.in.tum.de/documentation.html, June 2019.

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

[16] 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.

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

[18] Markus Wenzel. *Isabelle/Isar – a versatile environment for human-readable formal proof documents*. PhD thesis, Institüt für Informatik, Technische Universität München, 2002.

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

A. R. Martini
Av. Marechal Andrea 11/210
91340-400 - Porto Alegre - Brazil
e-mail: alfio.martini@gmail.com