Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 27, Num. 3 (2020) 84-101

RESEARCH ARTICLE

Reasoning about Partial Correctness Assertions in ISABELLE/HOL

Raciocinando sobre Asserções de Correção Parcial em ISABELLE/HOL

A. R. Martini[1]*

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 — Program verification — Theorem proving — Isabelle/HOL

Resumo: A lógica de Hoare tem uma longa tradição em verificação formal e tem sido continuamente desenvolvida e utilizada para verificar uma ampla classe de programas, incluindo programas sequenciais, orientados a objetos e concorrentes. O objetivo deste trabalho é fornecer uma exposição detalhada e acessível das várias maneiras pelas quais o usuário pode conduzir, explorar e escrever provas de correção de programas imperativos com a lógica de Hoare e o assistente de provas ISABELLE. Com a linguagem *Isar*, é possível escrever provas estruturadas e legíveis, adequadas para a compreensão e comunicação humana.
Palavras-Chave: Lógica de Hoare — Verificação de programas — Prova de teoremas — Isabelle/HOL

[1] *Av. Marechal Andrea, 11/210, Porto Alegre-RS-Brasil*
*__Corresponding author__: alfio.martini@gmail.com
DOI: https://doi.org/10.22456/2175-2745.98483 • **Received:** 25/11/2019 • **Accepted:** 28/04/2020

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 (undesired manipulation of shared variables), deadlock freedom and fair behavior are also important.

Using Floyd/Hoare logic [1, 2], 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 an 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 as-

sertion 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, 4].

On the other hand, modern proof assistants are 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. Interactive proof assistants like ISABELE/HOL [5] and COQ [6, 7], provide ways to specify and prove programs using Hoare logic with a great degree of automation. Moreover, Hoare Logic lies at the core of a multitude of tools that are being used in academia and industry to specify and verify real software systems. In fact, the current state of the art proof technology for programming languages and verification systems is based or inspired by Hoare logic [8, 9, 10].

In ISABELLE/HOL, the program semantics and the proof system 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 entailed by 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 work is to provide a detailed and accessible exposition of the several ways that one can conduct, explore and write proofs of correctness of sequential imperative programs with Hoare logic and the ISABELLE proof assistant. Besides that, we highlight a proof methodology 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 of a 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 with a simple arithmetic problem: the exponentiation operation with positive integer exponents. Section 5 discusses in detail a preliminary case study: reversal of polymorphic lists. In section 6 we discuss a more substantial case study, insertion sort, and discuss the need to prove several auxiliary lemmas to achieve almost full automation. In section 7 we discuss how to reason with Hoare logic in the proof assistant COQ and in the verification aware programming language DAPHNY. Finally, we summarize the main ideas and contributions of this work. The ISABELLE theory related to this development 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. We use “iff” as

an abbreviation for "if and only if".

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 and c is a program fragment in an 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 ***Var*** of program variables, ranged over by metavariables $x, y, \ldots$. The abstract syntax of the syntactic category ***Prg*** is given by the following grammar, where c_0, c_1, c range over ***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 [11].

$$\begin{array}{rcl} c \in \textbf{\textit{Prg}} & ::= & \underline{\textbf{skip}} \mid x := a \mid c_0; c_1 \\ & & \mid \underline{\textbf{while}}\ b\ \underline{\textbf{do}}\ c\ \underline{\textbf{od}} \\ & & \mid \underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ c_0\ \underline{\textbf{else}}\ c_1\ \underline{\textbf{fi}} \end{array}$$

The grammar presented above defines an imperative programming language that is simple enough for us to introduce in a precise way the main ideas underlying the semantics and proof theory of Hoare Logic. However, in some examples, using the same programming constructions, we will write programs that express computations not only over integers, but also over polymorphic types, like lists.

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 c 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 [12], we have the following partial correctness assertion to compute A^B for 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 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 $env : \textbf{\textit{IVar}} \to \mathbb{Z}$, where ***IVar*** is a countably infinite set of integer logical variables. The set of all such environments is $Env = (\textbf{\textit{IVar}} \to \mathbb{Z})$.

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: $\sigma \in \Sigma = (\textbf{\textit{Var}} \to \mathbb{Z})$. 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 $\sigma : \textbf{\textit{Var}} \to \mathbb{Z}$ with respect to the environment $env : \textbf{\textit{IVar}} \to \mathbb{Z}$.

A program assertion P is *valid* in an environment $env : \textbf{\textit{IVar}} \to \mathbb{Z}$, 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 $\langle \sigma, c \rangle \Downarrow_p \sigma'$ we mean that the execution of program c from the initial state σ leads to the final state σ' (see e.g, [11, 13]).

We say that a triple $\{P\}\ c\ \{Q\}$ is *true* at a state $\sigma \in \Sigma$ and environment $env : \boldsymbol{IVar} \to \mathbb{Z}$, 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* $env : \boldsymbol{IVar} \to \mathbb{Z}$, written $\models_{env} \{P\}\ c\ \{Q\}$ iff

$$\forall \sigma \in \Sigma.\ \sigma \models_{env} \{P\}\ c\ \{Q\}.$$

Finally, a partial correctness assertion (Hoare triple) is *(arithmetic) valid*, written $\models \{P\}\ c\ \{Q\}$, iff

$$\forall\, env : \boldsymbol{IVar} \to \mathbb{Z}.\ \models_{env} \{P\}\ c\ \{Q\}.$$

Note that all these semantic concepts are formulas 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 of provability, denoted by the symbol $\vdash$. 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 free occurrence of the program variable x in the assertion Q by the arithmetic expression a.

$$\frac{}{\vdash \{P\}\ \underline{\textbf{skip}}\ \{P\}}\ \textsf{Skip}$$

$$\frac{}{\vdash \{Q[x/a]\}\ x := a\ \{Q\}}\ \textsf{Ass}$$

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

$$\frac{\vdash \{P \wedge B\}\ c_1\ \{Q\} \quad \vdash \{P \wedge \neg B\}\ c_2\ \{Q\}}{\vdash \{P\}\ \underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ c_1\ \ \underline{\textbf{else}}\ c_2\ \underline{\textbf{fi}}\ \{Q\}}\ \textsf{IfE}$$

$$\frac{\vdash \{P \wedge B\} c \{P\}}{\vdash \{P\}\ \underline{\textbf{while}}\ b\ \underline{\textbf{do}}\ c\ \underline{\textbf{od}}\ \{P \wedge \neg B\}}\ \textsf{PWh}$$

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

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

The power of Floyd/Hoare treatment of imperative programs [2, 1] 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 an 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. 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*

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

$$\dfrac{\dfrac{\dfrac{\vdots\ \boxed{2}}{\vdash \{INV \wedge bw\}\ body\ \{INV\}}}{\vdash \{INV\}\ w\ \{INV \wedge \neg bw\}} \quad \vdots\ \boxed{3}}{\vdash \{INV\}\ w\ \{Pos\}}$$

$$\dfrac{\dfrac{\vdots\ \boxed{1}}{\vdash \{Pre\}\ init\ \{INV\}} \quad \dfrac{\vdots}{\vdash \{INV\}\ w\ \{Pos\}}}{\vdash \{Pre\}\ init; w\ \{Pos\}}$$

where the missing proof trees $\boxed{1}$, $\boxed{2}$ *and* $\boxed{3}$ *correspond, respectively, to the proof trees of the following proof obligations:*

$\boxed{1}$ *The invariant is true after initialization.*

$\boxed{2}$ *The invariant is preserved by the loop.*

$\boxed{3}$ *The invariant is strong enough to entail the postcondition, i.e.,* $\vdash INV \wedge \neg bw \rightarrow Pos$.

□

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 the proof tree $\boxed{2}$, where $AB \triangleq a = A \wedge b = B$ and $IAB \triangleq i + 1 \leq b \wedge AB, pIAB \triangleq p = a^{i+1} \wedge IAB, paIAB \triangleq p * a = a^{i+1} \wedge IAB$:

1 $\{pIAB\}\ i := i + 1\ \{INV\}$

2 $\{paIAB\}\ p := p * a\ \{pIAB\}$

3 $\{paIAB\}\ p := p * a; i := i + 1\ \{INV\}$

4 $\quad INV \wedge bw$

5 $\quad \vdots$

6 $\quad p * a = a^{i+1} \wedge IAB$

Line 1 and 2 follows from the assignment rule, and line 3 by the rule of composition. The implication outlined in the proof box $4-6$ could be proved, for instance, like this:

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

7 $INV \wedge bw \rightarrow paIAB$

where *Th* denote basic laws of arithmetic (theorems). Thus, to prove the original triple, it is sufficient to prove the following implications, also called *verification conditions* (see section 3.2):

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

These assertions are arithmetic valid and also provable in any reasonable presentation of the theory of integer arithmetic [14, 15].

3. On Automation of Hoare Logic

In this 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 $\{P\}\ c\ \{Q\}$, one generates subgoals, subsubgoals, etc., until the problem is solved.

A typical system 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 2.2 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.

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

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

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. An annotated command is a command with assertions inserted within it. A command is said to be properly annotated if assertions have been inserted at the following places:

- Before each command c_i (where $i > 0$) in a sequence $c_1; c_2; \ldots; c_m$ where each c_i is not an assignment command.
- After the word **do** in the **while** command.

Intuitively, the inserted assertions should express the conditions one expects to hold whenever control reaches the point at which the assertion occurs. A *properly annotated specification* is a Hoare triple $\{P\}\ c\ \{Q\}$ where c is a properly annotated command. Thus, the syntactic set of annotated command is defined by the following grammar:

$$\begin{array}{rl} c\ ::= & \underline{\textbf{skip}} \mid x := a \mid c_0; x := a \mid c_0; \{D\} c_1 \\ & \mid \underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ c_1\ \underline{\textbf{else}}\ c_2\ \underline{\textbf{fi}} \\ & \mid \underline{\textbf{while}}\ b\ \underline{\textbf{do}}\ \{D\} c\ \underline{\textbf{od}} \end{array}$$

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 **while** b **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* $INV \triangleq p = a^i \wedge i \leq b \wedge a = A \wedge b = B$.

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

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.

Definition 3.2. *The function that maps an annotated Hoare triple to its set of verifications conditions is defined by structural induction on annotated commands as follows:*

$$
\begin{array}{l}
vc(\{P\}\ \underline{\textbf{\textit{skip}}}\ \{Q\}) = \{P \rightarrow Q\} \\
vc(\{P\}\ x := a\ \{Q\}) = \{P \rightarrow Q[x/a]\} \\
vc(\{P\}\ c_0; x := a\ \{Q\}) = vc(\{P\}\ c_0\ \{Q[x/a]\}) \\
vc(\{P\}\ c_0; \{D\}\ c_1\ \{Q\}) = vc(\{P\}\ c_0\ \{D\}) \\
\cup\ vc(\{D\}\ c_1\ \{Q\}) \\
\qquad\qquad (c_1 \textit{ not an assignment}) \\
vc(\{P\}\ \underline{\textbf{\textit{if}}}\ b\ \underline{\textbf{\textit{then}}}\ c_1\ \ \underline{\textbf{\textit{else}}}\ c_2\ \underline{\textbf{\textit{fi}}}\ \{Q\}) \\
= vc(\{P \wedge b\}\ c_1\ \{Q\}) \\
\cup\ vc(\{P \wedge \neg b\}\ c_2\ \{Q\}) \\
vc(\{P\}\ \underline{\textbf{\textit{while}}}\ b\ \underline{\textbf{\textit{do}}}\ \{D\}c\ \underline{\textbf{\textit{od}}}\ \{Q\}) \\
= vc(\{D \wedge b\}\ c\ \{D\}) \\
\quad \cup \{P \rightarrow D\} \cup \{D \wedge \neg b \rightarrow Q\}
\end{array}
$$

□

Example 3.3. *Using definition 3.2 on the example 3.1, we have:*

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

□

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 [16]).

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 [1, 2], we can prove that a program is correct by applying a finite set of

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

inference rules to an initial program specification of the form $\{P\}\ c\ \{Q\}$ such that P and Q are logical assertions, and c is an 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. It is a generic proof assistant that allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. ISABELLE/HOL is the specialization of ISABELLE for HOL, which stands for *Higher Order Logic* [17]. We use the two terms interchangeably to denote specialization of ISABELLE to Higher Order Logic. HOL can be understood by the equation *HOL = Functional Programming + Logic*. Thus, most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight the essential notation. The space of total functions is denoted by the infix $\Rightarrow$. 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.

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

Figure 1. Power Algorithm in ISABELLE

Example 4.1. *The example 2.1 that computes the positive power of an integer is formalized in the library `HOL-Hoare` as shown in Figure 1.* □

ISABELLE automatically computes the type of each variable in a term. Despite type inference, it is sometimes necessary to attach explicit **type constraints** to a variable or term. The syntax is `t` :: τ as in `n` :: `nat`. In our example, we have added type constraints just to improve readability. The sequence of `apply` commands right after the end of the triple is called a *proof script*. It is a procedural low level language that is very useful to explore initial proof attempts, especially when the proof depends on a number of additional lemmas. Proof scripts is a document model for unstructured proofs and can only be understood if you are playing the script inside ISABELLE.

After applying the tactic `vcg`, the proof state in Figure 2 shows the three verification conditions that

must be proved (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.

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

Figure 2. Power Algorithm in ISABELLE - VC's

In ISABELLE, we have two important meta-level operators: the universal quantifier $\bigwedge$ and the implication $\Longrightarrow$. They are part of the ISABELLE framework, not of the logic HOL. They are used essentially for generality (to express the 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.

A structured presentation of the reasoning behind the validity of the second verification condition is formalized with the proof language Isar in Figure 3.

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 of Isar proofs: `fix` is used to declare arbitrary variables, `assume` to state the assumptions and `show` to assert the goal of the proof. To improve readability, we introduced an abbreviation, `?INV` for the invariant, after the `fix` variable declarations. The innermost `proof`...`qed` delimiters enclose the actual proof. 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 invariant into subformulas. The subsequent lines use the simplifier to prove basic arithmetical facts. The predefined name `this` can be used to refer to the proposition proved in the previous step. In the last step, the unknown `?thesis` is matched against the last declared `show`.The last `show` statement proves the actual goal. The `auto` after the outermost `qed` proves automatically the remaining verification conditions (the first and third).

```
proof (vcg)
  fix a b p i
  let ?INV = "p = a ^ i ∧ i ≤ b ∧ a = A ∧ b = B"
  assume ass:"?INV  ∧ i < b"
  show "p * a = a ^ (i + 1) ∧ i + 1 ≤ b
        ∧ a = A ∧ b = B"
    proof -
      from ass obtain 1: "p = a ^ i" and 2:"i ≤ b"
      and 3:"a = A ∧ b = B" and 5:"i<b"  by blast
      from 1 have 6:"p*a=a^i*a" by simp
      from this have 7:"p * a = a ^ (i + 1)" by simp
      from 5 have 8:"i+1≤b" by simp
      from this 3 7 show ?thesis by simp
    qed
qed (auto)
```

Figure 3. Isar proof of power's second VC

5. Imperative List Reversal

We can write an imperative program to reverse a list by repeatedly taking the head of the list and left appending it to an auxiliary variable that continuously stores a increasingly bigger prefix of the input list in a reverse way. Figure 4 presents a Hoare triple for a program that computes the reversal of a list according to this idea.

In the precondition, the logical variable `X` is used to save the initial value of the program variable

```
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 4. Hoare Triple - List Reversal

xs. In the loop, a new value of the accumulator is set with the head of the list, left 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 xs.
       xs = X ⟹
       rev xs @ [] = rev X
 2. ⋀acc xs.
       rev xs @ acc = rev X ∧
       xs ≠ [] ⟹
       rev (tl xs) @ hd xs # acc =
       rev X
 3. ⋀acc xs.
       rev xs @ acc = rev X ∧
       ¬ xs ≠ [] ⟹
       acc = rev X
```

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 second follows from the fact that $\texttt{xs} \neq [\,] \rightarrow \texttt{xs} = [\texttt{hd xs}] @ \texttt{tl xs}$ and the identity $\texttt{rev}\,(\texttt{as@bs}) = \texttt{rev bs@rev as}$. Using the isar proof language, we show a detailed proof of the second verification condition of the imperative list reversal in Figure 5.

```
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 5. Structured Proof of 2nd VC

The proof structure is analogous to the one presented in Figure 3. The proof itself is enclosed within the innermost `proof-qed` delimiters. 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 step follows by the theorem `rev(xs@ys) = (rev ys)@(rev xs)`. The fifth step follows 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).

6. Case Study: Insertion Sort

In this section we discuss the validity of a partial correctness assertion for the insertion sort algorithm. In the proof triples from Figures 1 and 4, all the auxiliary functions we needed to express the algorithm were already defined in ISABELLE's standard mathematical theory (called `Main`). For instance, we used arithmetic functions like addition and multiplication for the computation of the positive integer power. Also, the imperative list reversal algorithm was written with functions for computing the head and tail of a list, as well as a function for left appending an element to a list. The theory `Main` also provides a number of lemmas that we used in those proofs. Proof tactics like `simp` and `auto` apply them automatically. During the explanations, I have made explicit which lemmas were needed for those proofs to succeed.

For the insertion sort algorithm, besides basic facts about lists, which are already available to us in `Main`, we also need to define a few other ones, as well as prove essential relationships amongst them. We first introduce some auxiliary functions and related lemmas before we proceed with the proof of the triple itself. These functions are defined with the functional language of ISABELLE, which resembles a lot ML [18] and Haskell [19]. In Figure 6 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 6. Functions for Insertion Sort

The function `ins` inserts an element at the right position in a ordered list. Several of these function declarations have 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` [19, 20]. The funcion `iSort` returns a sorted list by repeated application of the function `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`. The `isorted` function states that 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 7. 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 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 7. 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 8 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 8. Insertion Sort Lemmas

The informal meaning of these lemmas can be understood as follows:`le_ins` states if a certain value precedes all elements of a list and also precedes another value `a`, then it also precedes all the elements of the list which includes `a`. Lemma `le_mon` says that the function $\lambda w.\ le\ w\ xs$ is monotonic w.r.t. to the order relation. Proposition `ins_sorted` asserts that the insert function preserves sortedness, while `is_sorted` claims that insertion sort always returns a sorted list. Lemma `ins_count` asserts that counting is compatible with insertion of new elements, and `count_sum` proves that counting the number of occurrences is compatible with concatenation of lists. Proposition `len_sort` says that the length of an input list is invariant under insertion sort, while `count_isort` affirm that the number of occurrences of elements is invariant under sorting. Finally, `ins_len` states that the lenght of list is compatible with insertion of new elements.

ISABELLE's `auto` 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 [5], chapter 2. After calling the verification condition generator we are left with the following proof goals:

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

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. Because two of the three verification conditions are trivial, we can 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.

We introduce here another ISABELLE powerful too for theorem proving, called SLEDGEHAMMER. SLEDGEHAMMER [21] 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 like E, SPASS and vampire [22, 23, 24] and *SMT* solvers like CVC4 and Z3 [25, 26].

A proof script that solves the three goals is shown bellow.

```
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}"
 apply (vcg)
 apply (auto simp add:is_perm_def) — ‹ 1 ›
   apply (simp add: ins_sorted) — ‹1.1›
   apply (simp add: ins_len) — ‹1.2›
   apply  (smt count.simps(2) count_sum
          ins_count list.collapse) —‹1.3›
done
```

Because two of the three verification conditions are trivial, we try to prove everything automatically applying `(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` discovered by SLEDGEHAMMER (lines 1.1, 1.2 and 1.3). 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. 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 structured proof should always be the final, polished documentation of a formally verified reasoning in ISABELLE.

```
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 9. Insertion Sort - Proof Draft

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 9. 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 must later be refined with

a real proof. It is a fake proof pretending to solve the pending claim without further ado.

We must be very careful, though: every time we use `sorry`, we are leaving a door open for total nonsense to enter ISABELLE's nice, rigorous, formally checked environment! However, if used wisely, this approach 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.

```
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)
    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 10. Insertion Sort - Structured Proof

In Figure 10 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, it composes the theorems `calculation` and `this` (i.e. the two previous equations) using transitivity of equality. The command `finally` is a shorthand for `also from calculation`. Thus, in the `finally` step, the chain of equations is closed with respect to the transitivity rule and it is stored in the variable `calculation`. The unknown `?thesis` is implicitly matched against the assertion stated by the previous `have` command.

7. Verification Tools for Hoare Logic & Programs with Assertions

In section 1 I have referenced a number of interactive proof assistants and verification tools with which one can mechanize and automate reasoning about Hoare Logic and, in more generality, about imperative programs with assertions. Here I include a brief discussion about two of these related approaches.

7.1 Coq

COQ [6, 7] is an interactive theorem prover along the same lines as ISABELLE. It allows the expression of mathematical assertions, mechanically checks proofs of these assertions, helps to find formal proofs, and extracts a certified program from the constructive proof of its formal specification. A solid implementation of Hoare Logic in COQ can be found in the first two volumes of the classic *Software Foundation* series [27, 28]. The development

is analogous to the one done in ISABELLE:

1. Formalization of the programming language syntax.
2. Formalization of the programming language semantics via an inductive relation (big-step operational semantics).
3. Formalization of proof rules as theorems with respect to the semantics.
4. Computation of verification conditions of annotated programs
5. Proof of verification conditions using interactive or automatic tactics (proof procedures).

As an example, consider the simple algorithm for slow subtraction bellow, taken from [27]:

```
Example subtract_slowly_dec (m : nat) (p : nat)
 : decorated :=
    {{ fun st => st X = m /\ st Z = p }} ->>
    {{ fun st => st Z - st X = p - m }}
  WHILE ~(X = 0)
  DO   {{ fun st => st Z - st X = p - m
        /\ st X <> 0 }} ->>
       {{ fun st => (st Z - 1) -
        (st X - 1) = p - m }}
     Z ::= Z - 1
       {{ fun st => st Z - (st X - 1) = p - m }};;
     X ::= X - 1
       {{ fun st => st Z - st X = p - m }}
  END
    {{ fun st => st Z - st X = p - m
     /\ st X = 0 }} ->>
    {{ fun st => st Z = p - m }}.

Theorem subtract_slowly_dec_correct : forall m p,
  dec_correct (subtract_slowly_dec m p).
Proof. intros m p. verify. Qed.
```

In this formalization of Hoare logic we have some limitations:

- Reasoning is limited to imperative programs that compute over natural numbers.
- The programs must be *decorated*. Concretely, a decorated program consists of the program text interleaved with assertions (either a single assertion or possibly two assertions separated by an implication).
- Assertions must be encoded as semantic functions using lambda abstractions.

As a comparison, the same solution written in ISABELLE is shown below:

```
lemma slow_sub_isa:
"VARS  (x::nat) (z::nat)
{x=M ∧ z=P}
WHILE ¬(x=0)
  INV { z-x = P-M ∧ x≥0}
  DO z:=z-1;x:=x-1 OD
{z = P-M}"
apply (vcg)
apply (auto)
done
```

7.2 DAPHNY

DAFNY is an automated verification aware programming language meant to aid software engineers in the design of correct programs. Behind the scenes, DAFNY first converts code into the mathematical expressions of Hoare logic with the aid of a intermediate language called Boogie [29], and then it sends the code to the theorem prover Z3 [25]. Two of the most fundamental annotations in DAFNY are the preconditions and postconditions. In the programming language, preconditions are expressed with the keyword *requires*, and postconditions with the keyword *ensures*. Another fundamental annotation in DAFNY is the loop invariant. Loop invariants, given with an *invariant* statement, are expressions that must hold true before the initialization of the loop, throughout all iterations, and upon termination. At the end of the loop, invariants also provide information that assist in verifying the overall method and the postconditions. Invariants are often restatements of the postconditions in terms of intermediate variables, and by outlining these steps, DAFNY is able to verify that these statements which hold constant for the duration of the loop, imply the postcondition. DAFNY also proves program termination, i.e., that it does not loop forever, by using *decreases* annotations to formalize *loop variants*. Loop variants are athematical functions defined on the state space of a computer program whose value is monotonically decreased with respect to a well-founded relation by the iteration of a while loop under some invariant conditions. The *slow subtraction* example introduced above can be formalized in DAFNY as in Figure 11:

```
method slow_sub_daphny(m:int, p:int)
returns (z:int)
        requires m >=0 && p >= 0
        ensures z == p - m;
    {
        var x:int := m;
        z := p;
        while !(x==0)
        decreases x
        invariant z - x == p - m
        invariant x >= 0
        {
            z := z - 1;
            x := x - 1;
        }
    }
```

Figure 11. Slow subtraction in Daphny

DAFNY offers features from both imperative programming (assignments, loops, and classes with dynamically allocated instances) and functional programming (algebraic datatypes and functions). The DAFNY verifier runs continuously and whenever it cannot verify a proof obligation, it flags it as an error, much like a word processor immediately marks dubious spelling or grammar.

Author contributions

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.

The implementation of Hoare logic that comes with the standard distribution of the ISABELLE provides a standard syntax to declare Hoare triples and supports a varied set of proof tactics for proof construction and automation. Moreover, the user can write programs over a rich collection of data types which are formalized in higher order logic.

In this article I have tried to convey a detailed and accessible presentation to several techniques available for reasoning with Hoare Logic in the proof assistant ISABELLE within a certain proof methodology:

1. Construction of proofs using procedural proof scripts that discharge the verification conditions using the several automatic proof procedures available in ISABELLE: `simp`, `auto`, `force`, `blast` and `fastforce`.

2. If the previous step fails to discharge some of the verification conditions, the user can invoke SLEDGEHAMMER to see if it discovers a proof for each undischarged verification condition.

3. Above all, is strongly advised writing a structured proof using the proof language *Isar*, using intermediate steps and auxiliary lemmas as needed, in order to break down the complexity of the task.

In the last step, a proof draft like the one in Figure 9 can be very adequate as an proof scheme or proof draft. It is also helpful to realize that new auxiliary lemmas must be stated and proved, e.g. like the ones in Figure 8. Using the proof tactic `auto` with proof scripts is always a effective approach to realize what are the missing theorems.

Despite ISABELLE's varied and sophisticated tools for proof automation, 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.

Therefore, I have emphasized through all the examples in the paper, that the use of the proof language `Isar` for documentation, reasoning, and especially communication at a high-level, is essential. A structured proof should always be the final, polished documentation of a formally verified reasoning in ISABELLE.

References

[1] FLOYD, R. W. Assigning meanings to programs. *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, v. 19, 01 1967.

[2] HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, v. 12, n. 10, p. 576–580, 1969.

[3] APT, K. R.; BOER, F. de; OLDEROG, E.-R. *Verification of Sequential and Concurrent Programs*. 3rd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009.

[4] ANDREWS, G. R. *Concurrent Programming: Principles and Practice*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991.

[5] NIPKOW, T.; WENZEL, M.; PAULSON, L. C. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[6] APPEL, A. W. et al. *Program Logics for Certified Compilers*. New York, NY, USA: Cambridge University Press, 2014.

[7] BERTOT, Y.; CASTRAN, P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010.

[8] FILLIÂTRE, J.-C. One Logic To Use Them All. In: BONACINA, M. P. (Ed.). *CADE 24 - the 24th International Conference on Automated Deduction*. Lake Placid, NY, United States: Springer, 2013.

[9] LEINO, R. Dafny: An automatic program verifier for functional correctness. In: *16th International Conference, LPAR-16, Dakar, Senegal*. [S.l.]: Springer Berlin Heidelberg, 2010. p. 348–370.

[10]BUBEL, R.; HÄHNLE, R. Key-hoare. In: *Deductive Software Verification - The KeY Book - From Theory to Practice*. [S.l.: s.n.], 2016. p. 571–589.

[11]WINSKEL, G. *The Formal Semantics of Programming Languages - an Introduction*. [S.l.]: MIT Press, 1993. I-XVIII, 1-361 p. (Foundation of computing series).

[12]HEIN, J. *Discrete Structures, Logic, and Computability*. [S.l.]: Jones & Bartlett Learning, 2010.

[13]NIELSON, H. R.; NIELSON, F. *Semantics with Applications - a Formal Introduction*. [S.l.]: Wiley, 1992. I-XII, 1-240 p. (Wiley professional computing).

[14]BRADLEY, A. R.; MANNA, Z. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer-Verlag, 2007.

[15]LOECKX, J.; SIEBER, K.; STANSIFER, R. D. *The Foundations of Program Verification*. New York, NY, USA: John Wiley & Sons, Inc., 1984.

[16]GORDON, M. J. C. *Programming language theory and its implementation - applicative and imperative paradigms*. [S.l.]: Prentice Hall, 1988. (Prentice Hall International series in Computer Science).

[17]NIPKOW, T.; KLEIN, G. *Concrete Semantics: With Isabelle/HOL*. [S.l.]: Springer Publishing Company, Incorporated, 2014.

[18]PAULSON, L. C. *ML for the working programmer (2. ed.)*. [S.l.]: Cambridge University Press, 1996.

[19]O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. *Real world Haskell - code you can believe in*. O'Reilly, 2008. Disponível em: ⟨http://book.realworldhaskell.org/⟩.

[20]HAFTMANN, F. *Haskell-style type classes with Isabelle/Isar*. 2019. ⟨http://isabelle.in.tum.de/documentation.html⟩.

[21]BLANCHETE, J. *User's Guide to Sledgehammer*. 2018. ⟨http://isabelle.in.tum.de/documentation.html⟩.

[22]SCHULZ, S.; CRUANES, S.; VUKMIROVIC, P. Faster, higher, stronger: E 2.3. In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*. [s.n.], 2019. p. 495–507. Disponível em: ⟨https://doi.org/10.1007/978-3-030-29436-6_29⟩.

[23]WEIDENBACH, C. et al. Spass version 3.5. In: SCHMIDT, R. A. (Ed.). *CADE*. [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5663), p. 140–145.

[24]KOVÁCS, L.; VORONKOV, A. First-order theorem proving and vampire. In: *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*. New York, NY, USA: Springer-Verlag New York, Inc., 2013. (CAV 2013), p. 1–35.

[25]MOURA, L. D.; BJØRNER, N. Z3: An Efficient SMT Solver. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.]: Springer-Verlag, 2008. (TACAS'08/ETAPS'08), p. 337–340.

[26]BARRETT, C. et al. CVC4. In: GOPALAKRISHNAN, G.; QADEER, S. (Ed.). *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. [S.l.]: Springer, 2011. (Lecture Notes in Computer Science, v. 6806), p. 171–177.

[27]PIERCE, B. C. et al. *Logical Foundations*. [S.l.]: Electronic textbook, 2018. ⟨https://softwarefoundations.cis.upenn.edu/⟩. (Software Foundations series, volume 1).

[28]PIERCE, B. C. et al. *Programming Language Foundations*. [S.l.]: Electronic textbook, 2018. ⟨https://softwarefoundations.cis.upenn.edu/⟩. (Software Foundations series, volume 2).

[29]LEINO, K. R. M. This is boogie 2. 2008. Disponível em: ⟨https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/⟩.