Auxiliary Variables and Recursive Procedures

Thomas Schreiber

LFCS Edinburgh, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland

Abstract. Much research in axiomatic semantics suffers from a lack of formality. In particular, most proposed verification calculi for imperative programs dealing with recursive procedures are known to be unsound or incomplete. Focussing on total correctness, we present a new consequence rule which yields a sound and complete Hoare-style calculus in the presence of parameterless recursive procedures. Both, the standard consequence and an improved rule of adaptation are instances of our new rule. This work has been developed under the auspices of the computer-aided proof system LEGO. The rigorous treatment of auxiliary variables has been crucial for establishing our results. A comparison with VDM reinforces our view that auxiliary variables deserve to be treated seriously.

1 Introduction

What is a good framework for formally developing programs from specifications? Design criteria include notions of soundness and completeness. In this paper, focussing on total correctness, we investigate verification calculi for imperative programs with recursive procedures based on input/output specifications.

We present a new Hoare-style calculus and extend VDM's decomposition rules [15] in the context of recursive procedures, proving soundness and completeness for both systems under the auspices of the computer-aided proof system LEGO [16].

One of our aims is to demonstrate that it is not only feasible but easier to work on selected research areas using current proof assistants. Most published verification calculi for imperative programs dealing with recursive procedures are known to be either unsound or incomplete, despite authors backing up their claims with "proofs" [6]. No such proof attempts would have been accepted by a mechanical proof checker. Furthermore, we believe that in most cases, correct soundness and completeness proofs require little overhead when being done on a machine provided the area is formally understood. Previously, auxiliary variables in Hoare logic have been given insufficient attention. Apt and Meertens [4] have proposed a method for formally integrating auxiliary variables in assertions. We extend this idea to Hoare logic.

In the following section, we present design criteria for verification calculi. Hoare logic and VDM are investigated in the light of these requirements.

Section 3 introduces Hoare logic for simple imperative programs. This section contains no new results, it is merely intended to serve as a gentle introduction to developing imperative programs from input/output specifications.

Section 4 considers recursive procedures. Parameter passing is an orthogonal problem which, following Apt [3], we omit in this paper. We motivate a new consequence rule leading to an improved Hoare logic calculus for imperative programs with recursive procedures. A comparison with VDM reinforces our view that auxiliary variables deserve a rigorous treatment.

The symbol 🗎 indicates that a corresponding LEGO script is available on-line, point your Web browser to `http://www.dcs.ed.ac.uk/home/tms/lego/tapsoft97`. In this paper, we abstract from the details and present our results in a more conventional mathematical format. However, we need to occasionally rely on a more formal notation, closer to the actual LEGO scripts, to resolve ambiguities arising from informal presentations. For the reader familiar with standard techniques for mechanising programming logics [9, 18], the presentation of this paper is self-contained and provides sufficient information to exploit our work in other modern computer-aided proof systems such as Coq, HOL, Isabelle or PVS.

2 Design Criteria for Verification Calculi

Let $\mathcal{M}$ be a model interpreting constants, functions and relations of both the programming language with typical element S and a logical language *Pre* with typical element P. One can then extend the language *Pre* and its notion of validity $\mathcal{M} \models \mathcal{P}$ to correctness formulae S sat *Spec* relating specifications *Spec* and programs S. Validity of $\mathcal{M} \models S$ sat *Spec* is defined in terms of validity of the underlying logical language and the expected behaviour of programs (which we shall axiomatise via operational semantics). In the sequel, we omit the model $\mathcal{M}$, assuming implicitly that we are working with a standard model.

The logical languages used in practice are too expressive for model checking to be feasible in the context of sequential imperative programs. Furthermore, reasoning directly based on the underlying operational semantics is too clumsy. A verification calculus ought to provide a more abstract interface. To *implement* a computer-aided framework for developing correct programs from specifications, one needs to establish a verification calculus containing a set of axioms and rules for *deriving* proposition of the form $\vdash S$ sat *Spec*. What is the correspondence between $\models S$ sat *Spec* and $\vdash S$ sat *Spec*? Ideally, we would want that $\models S$ sat *Spec* if and only if $\vdash S$ sat *Spec*:

Definition 2.1 (Soundness). Only valid specifications can be derived i.e., $\vdash S$ sat *Spec* implies $\models S$ sat *Spec*.

Definition 2.2 (Completeness). All valid specifications can be derived i.e., $\models S$ sat *Spec* implies $\vdash S$ sat *Spec*.

Remark 2.3 (Relative completeness). If the underlying logical language *Pre* is too weak then $\vdash S$ sat *Spec* may not hold despite $\models S$ sat *Spec* because a refined specification required in the derivation of $\vdash S$ sat *Spec* cannot be expressed.

Conversely, for expressive logical languages such as Peano Arithmetic, a consistent formal system allowing one to infer all valid formulae cannot exist due to Gödel's incompleteness result. In particular, one cannot expect to achieve completeness for the larger class of correctness formulae S sat *Spec*.

To factor out problems concerning the underlying logical language, Cook [5] proposed that one investigates relative completeness: One should only consider sufficiently expressive logical languages. Furthermore, in defining a formal system for $\vdash S$ sat *Spec*, one may assume that all valid formulae of the underlying logical language are derivable i.e., $\models P$ implies $\vdash P$. We follow Cook's provisions respectively by restricting our attention to intuitionistic higher-order logic[1] and instead of assuming completeness of *Pre*, we define $\models S$ sat *Spec* relative to provability $\vdash P$ rather than to validity $\models P$ of the underlying logical language. This is a standard technique in mechanising programming logics because provability is a primitive concept in interactive proof systems.

3 Imperative Programs without Procedures

In this section, we restrict ourselves to basic language features, the empty statement, assignment, sequential composition, conditional and loop:

Definition 3.1 (Syntax). Imperative programs S: prog are defined by the following BNF grammar:

$$S ::= \textbf{skip} \mid x := t \mid S_1 ; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S$$

We need to *axiomatise* the intended behaviour of programs. Formal verification is relative to this axiomatisation and independent of specific compilers and hardware.

The behaviour of an imperative program depends in general on the contents of the memory. A particular snapshot of the memory is called a state. For the restricted class of programs considered in this paper, it suffices to model the set of all states Σ as the function space from program variable names to values.

Structural operational semantics provides a clean way to specify the effect of each language constructor in an arbitrary state:

Definition 3.2 (Semantics). The operational semantics is defined as the least relation $. \xrightarrow{\cdot} . : \Sigma \times \texttt{prog} \times \Sigma \to \texttt{Prop}$ satisfying

$$\sigma \xrightarrow{\textbf{skip}} \sigma$$

$$\sigma \xrightarrow{x := t} \sigma[x \mapsto t]$$

$$\frac{\sigma \xrightarrow{S_1} \eta \quad \eta \xrightarrow{S_2} \tau}{\sigma \xrightarrow{S_1 ; S_2} \tau}$$

[1] the internal logic of the LEGO system

$$\frac{\sigma \xrightarrow{S_1} \tau}{\sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2} \tau} \quad \text{provided } b(\sigma) \ .$$

$$\frac{\sigma \xrightarrow{S_2} \tau}{\sigma \xrightarrow{\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2} \tau} \quad \text{provided } \neg b(\sigma) \ .$$

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \sigma \quad \text{provided } \neg b(\sigma) \ .$$

$$\frac{\sigma \xrightarrow{S} \eta \quad \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau}{\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau} \quad \text{provided } b(\sigma) \ .$$

The kind Prop is the type of propositions in intuitionistic higher-order logic. In the context of a programming language, a boolean expression b may refer to the value of program variables. We have modelled boolean expressions as boolean valued functions of state $b: \Sigma \to \texttt{bool}$.

Assertions are propositions possibly containing references to program variables. We model assertions as predicates on states $\Sigma \to \texttt{Prop}$. We lift propositions point-wise to assertions and overload notation e.g. for an assertion $p: \Sigma \to \texttt{Prop}$ and a boolean expression $b: \Sigma \to \texttt{bool}$, the expression $p \wedge b$ is represented by $\lambda\sigma: \Sigma p(\sigma) \wedge \texttt{is\_true}\,(b(\sigma))$ where `is_true` is the standard coercion from the boolean type `bool` to the type of propositions `Prop`. While intuitionistic higher-order logic turns out to be well-suited for the research presented in this paper, users of other proof assistants may prefer different assertion languages[2].

Definition 3.3 (The semantics of Hoare logic). For assertions p, q and program S, the specification schema

$$\models_{\text{Hoare}} \{\cdot\} \ \cdot \ \{\cdot\} \ : \ (\Sigma \to \texttt{Prop}) \times \texttt{prog} \times (\Sigma \to \texttt{Prop}) \to \texttt{Prop}$$

$$\models_{\text{Hoare}} \{p\} \ S \ \{q\} \triangleq \forall\sigma: \Sigma \cdot p(\sigma) \Rightarrow \exists\tau: \Sigma \cdot (\sigma \xrightarrow{S} \tau) \wedge q(\tau)$$

characterises Hoare logic for total correctness. It is valid if for all initial states σ satisfying the precondition p, the program S terminates in a final state τ such that the postcondition q holds.

When we want to show that a particular program S satisfies a specification, we can exploit the inductive definition of the operational semantics. However, in practice, this will be too tedious, because the operational semantics presentation is in general not sufficiently abstract. It is advisable to establish a set of axioms and rules for deriving correctness judgements.

Based on work of Floyd [8], Hoare [12] proposed a verification calculus (originally for partial correctness) now referred to as Hoare logic. The following presentation contains a refined loop rule due to Harel [11] which leads to total correctness.

[2] In classical systems i.e., in which the axiom of excluded middle holds, the distinction between the types `bool` and `Prop` is not required.

Definition 3.4. A verification calculus for Hoare logic is defined as the least relation $\vdash_{\mathbf{Hoare}} \{.\} \;.\; \{.\} : (\Sigma \to \mathtt{Prop}) \times \mathtt{prog} \times (\Sigma \to \mathtt{Prop}) \to \mathtt{Prop}$ satisfying

$$\vdash_{\mathbf{Hoare}} \{p\}\ \mathbf{skip}\ \{p\} \tag{1}$$

$$\vdash_{\mathbf{Hoare}} \{p[x \mapsto t]\}\ x := t\ \{p\} \tag{2}$$

$$\frac{\vdash_{\mathbf{Hoare}} \{p\}\ S_1\ \{r\} \qquad \vdash_{\mathbf{Hoare}} \{r\}\ S_2\ \{q\}}{\vdash_{\mathbf{Hoare}} \{p\}\ S_1; S_2\ \{q\}} \tag{3}$$

$$\frac{\vdash_{\mathbf{Hoare}} \{p \wedge b\}\ S_1\ \{q\} \qquad \vdash_{\mathbf{Hoare}} \{p \wedge \neg b\}\ S_2\ \{q\}}{\vdash_{\mathbf{Hoare}} \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}} \tag{4}$$

$$\frac{\forall n{:}\mathbb{N} \cdot \vdash_{\mathbf{Hoare}} \{p(n+1)\}\ S\ \{p(n)\}}{\vdash_{\mathbf{Hoare}} \{\exists n{:}\mathbb{N} \cdot p(n)\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{p(0)\}}$$
$$\text{provided } \forall \sigma, \tau{:}\Sigma \cdot \forall n{:}\mathbb{N} \cdot (p(n+1)(\sigma) \Rightarrow b(\sigma)) \wedge (p(0)(\tau) \Rightarrow \neg b(\tau)) \tag{5}$$

Theorem 3.5 (Soundness). *The above verification calculus is sound.*

Proof. □

Completeness It is easy to see that the above Hoare calculus cannot be complete. Assuming completeness, we can show that checking the correctness of a program would be decidable: There is one rule for every constructor of the programming language. Given an arbitrary specification $\vdash_{\mathbf{Hoare}} \{p\}\ S\ \{q\}$, it suffices to check if the assertions p, q match the assertions in the conclusion of the rule corresponding to the structure of the program S. If not, $\vdash_{\mathbf{Hoare}} \{p\}\ S\ \{q\}$ is not derivable. Otherwise, we recursively examine the premises of the applied rule. This process either terminates in a rule being rejected or with no premises. In the latter case, the program S satisfies the specification $\vdash_{\mathbf{Hoare}} \{p\}\ S\ \{q\}$.

To obtain completeness, we must be able to equivalently transform assertions or, in particular in the case of loops, weaken the precondition and strengthen the postcondition [10]. Adding the *consequence rule*

$$\frac{\vdash_{\mathbf{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\mathbf{Hoare}} \{p\}\ S\ \{q\}} \qquad \text{provided } \forall \sigma, \tau{:}\Sigma \cdot p(\sigma) \Rightarrow p_1(\sigma) \wedge q_1(\tau) \Rightarrow q(\tau) \tag{6}$$

leads to a complete system while retaining soundness:

Lemma 3.6 (Soundness). *The consequence rule (6) preserves soundness.*

Proof. Straightforward.

Theorem 3.7 (Completeness). *The verification calculus defined as the least relation satisfying (1)–(6) is complete.*

Proof. See [11]. In the context of partial correctness, a completeness proof has recently been mechanised in Isabelle by Nipkow [18].

4 Imperative Programs with Recursive Procedures

In this section, we extend Hoare logic for recursive procedures. Parameter passing is an *orthogonal* issue which, following Apt [3], we omit in this paper. For simplicity, we also restrict our attention to the case of a single procedure declaration. We expect no difficulties in generalising the results in this paper to mutually recursive procedures. In the sequel, S_0: prog denotes the body of the procedure.

Definition 4.1 (Syntax). We extend the syntax by the constructor **call** which ought to invoke the body of the procedure, S_0.

$$S ::= \textbf{skip} \mid x := t \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S \mid \textbf{call}$$

Example 4.2 (Procedure declaration). A recursive procedure declaration for computing the factorial function $x!$ is given by

$$\begin{aligned} S_0 \triangleq\ & \textbf{if } x = 0 \textbf{ then } y := 1 \\ & \qquad\qquad \textbf{else } \ x := x - 1; \textbf{call}; x := x + 1; y := y * x \\ & \textbf{fi} \end{aligned}$$

Definition 4.3 (Semantics). A procedure call results in executing the body of the procedure S_0. We extend the operational semantics from Definition 3.2 by

$$\frac{\sigma \xrightarrow{S_0} \tau}{\sigma \xrightarrow{\textbf{call}} \tau}$$

Sokołowski [20] has proposed the rule

$$\frac{\forall n{:}\mathbb{N} \cdot \{p(n)\}\ \textbf{call}\ \{q\} \vdash_{\text{Hoare}} \{p(n+1)\}\ S_0\ \{q\}}{\emptyset \vdash_{\text{Hoare}} \{\exists n{:}\mathbb{N} \cdot p(n)\}\ \textbf{call}\ \{q\}} \quad \text{provided } \forall \sigma{:}\Sigma \cdot \neg p(0)(\sigma) \qquad (7)$$

to deal with recursive procedures. For a procedure call to terminate, there must be a finite recursive depth n. For $n = 0$, we have no recursion, for $n+1$, it suffices to show that the procedure body satisfies the specification. In this derivation, we may employ the original correctness formula as an additional *axiom*. However, the assumed Hoare triple must have recursive depth n. A simple form of contexts can capture such an additional assumption:

Definition 4.4 (Context). Contexts contain at most one correctness formula for procedure calls. A context Γ: `Context` can be represented by the BNF grammar $\Gamma ::= \emptyset \mid \{p\}\ \textbf{call}\ \{q\}$.

Hoare triples are from now on annotated by contexts i.e., $. \vdash_{\text{Hoare}} \{\cdot\} \cdot \{\cdot\}$: `Context` $\times$ ($\Sigma \to$ `Prop`) $\times$ prog $\times$ ($\Sigma \to$ `Prop`) $\to$ `Prop`. As usual, in the sequel, we omit the empty context $\emptyset$ in Hoare triples.

The previously given rules (1)–(6) need to be revised to support contexts:

$$\Gamma \vdash_{\textbf{Hoare}} \{p\}\ \textbf{skip}\ \{p\} \tag{8}$$

$$\Gamma \vdash_{\textbf{Hoare}} \{p[x \mapsto t]\}\ x := t\ \{p\} \tag{9}$$

$$\frac{\Gamma \vdash_{\textbf{Hoare}} \{p\}\ S_1\ \{r\} \quad \Gamma \vdash_{\textbf{Hoare}} \{r\}\ S_2\ \{q\}}{\Gamma \vdash_{\textbf{Hoare}} \{p\}\ S_1; S_2\ \{q\}} \tag{10}$$

$$\frac{\Gamma \vdash_{\textbf{Hoare}} \{p \wedge b\}\ S_1\ \{q\} \qquad \Gamma \vdash_{\textbf{Hoare}} \{p \wedge \neg b\}\ S_2\ \{q\}}{\Gamma \vdash_{\textbf{Hoare}} \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}} \tag{11}$$

$$\frac{\forall n{:}\mathbb{N} \cdot \Gamma \vdash_{\textbf{Hoare}} \{p(n+1)\}\ S\ \{p(n)\}}{\Gamma \vdash_{\textbf{Hoare}} \{\exists n{:}\mathbb{N} \cdot p(n)\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p(0)\}}$$
$$\text{provided } \forall \sigma, \tau{:}\Sigma \cdot \forall n{:}\mathbb{N} \cdot (p(n+1)(\sigma) \Rightarrow b(\sigma)) \wedge (p(0)(\tau) \Rightarrow \neg b(\tau)) \tag{12}$$

$$\frac{\Gamma \vdash_{\textbf{Hoare}} \{p_1\}\ S\ \{q_1\}}{\Gamma \vdash_{\textbf{Hoare}} \{p\}\ S\ \{q\}} \quad \text{provided } \forall \sigma, \tau{:}\Sigma \cdot p(\sigma) \Rightarrow p_1(\sigma) \wedge q_1(\tau) \Rightarrow q(\tau) \tag{13}$$

Furthermore, we need to add an axiom scheme

$$\{p\}\ \textbf{call}\ \{q\} \vdash_{\textbf{Hoare}} \{p\}\ \textbf{call}\ \{q\} \tag{14}$$

capturing the meaning of contexts.

Remark 4.5. Instead of introducing contexts, one could consider adding a higher-order variant of (7):

$$\frac{\forall n{:}\mathbb{N} \cdot \vdash_{\textbf{Hoare}} \{p(n)\}\ \textbf{call}\ \{q\} \Rightarrow \vdash_{\textbf{Hoare}} \{p(n+1)\}\ S_0\ \{q\}}{\vdash_{\textbf{Hoare}} \{\exists n{:}\mathbb{N} \cdot p(n)\}\ \textbf{call}\ \{q\}}$$
$$\text{provided } \forall \sigma{:}\Sigma \cdot \neg p(0)(\sigma) \tag{15}$$

As a drawback, (15) is not a valid constructor in an inductive definition of the relation $\vdash_{\textbf{Hoare}}$ due to the negative occurrence of $\vdash_{\textbf{Hoare}}$ in the premiss. One might also question the adequacy of this formulation: Instead of merely adding

$$\vdash_{\textbf{Hoare}} \{p(n)\}\ \textbf{call}\ \{q\} \tag{16}$$

as a new temporary *axiom* in deriving $\vdash_{\textbf{Hoare}} \{p(n+1)\}\ S_0\ \{q\}$, the premiss of (15) also permits induction on the *derivation* of (16).

Theorem 4.6 (Soundness). *The verification calculus defined by the least relation $\vdash_{\textbf{Hoare}}$ satisfying (7)–(14) is sound.*

Proof. By induction on the derivation of $\Gamma \vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$, we show simultaneously

1. $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ whenever $\Gamma = \emptyset$ and
2. $(\models_{\text{Hoare}} \{p_1\}\ \textbf{call}\ \{q_1\}) \Rightarrow (\models_{\text{Hoare}} \{p\}\ S\ \{q\})$ whenever $\Gamma = \{p_1\}\ \textbf{call}\ \{q_1\}$, reflecting the semantics of non-empty contexts.

□

However, the above presentation of Hoare logic catering for recursive procedures is not complete:

Example 4.7 (Incompleteness results). The procedure being previously declared in Example 4.2 correctly implements the factorial function. Using the axiomatic system, we can show that

$$\vdash_{\text{Hoare}} \{\mathbf{true}\}\ \textbf{call}\ \{y = x!\} \tag{17}$$

is derivable, but we cannot prove that the value of the program variable x remains invariant i.e., $\nvdash \{x = \mathbf{z}\}\ \textbf{call}\ \{x = \mathbf{z}\}$ where $\mathbf{z}$ is an auxiliary variable, see [3] for a proof. Unfortunately, this jeopardises the credibility of (17). The procedure declaration $S_0 \triangleq x := 1; y := 1$ would also satisfy (17) *without* leaving the variable x invariant.

4.1 A Better Consequence Rule

Auxiliary variables are to blame for incompleteness. They are usually considered as program variables or as (meta-) logical variables not occurring in programs, but they deserve a more rigorous treatment. Auxiliary variables are crucial in specifying properties. In Hoare logic where assertions are predicates on the initial and final state respectively, auxiliary variables are the only means to directly relate input and output. Almost every proper specification relies on auxiliary variables! It is inadequate to treat them as program variables or meta-logical variables. Otherwise, additional rules to achieve completeness tend to be somewhat elaborate to compensate for a too liberal notion of auxiliary variables: They must never occur in programs and, unless they appear in *both* pre- and postcondition, they can be eliminated by the consequence rule (13).

The role of auxiliary variables has been recognised by Apt and Meertens: For an arbitrary domain $\mathbf{T}$, assertions may depend on the value of program variables, characterised by the domain of states Σ and auxiliary variables, characterised by the domain $\mathbf{T}$ i.e. assertions can be considered as relations $\mathbf{T} \to \Sigma \to \texttt{Prop}$ [4]. We extend this idea to Hoare logic.

Definition 4.8 (Semantics of Hoare logic). For assertions $p, q \colon \mathbf{T} \to \Sigma \to \texttt{Prop}$ and programs $S \colon \texttt{prog}$, we can capture total correctness specifications incorporating auxiliary variables by the relation

$$\models_{\text{Hoare}} \{\cdot\}\ .\ \{\cdot\}\ :\ (\mathbf{T} \to \Sigma \to \texttt{Prop}) \times \texttt{prog} \times (\mathbf{T} \to \Sigma \to \texttt{Prop}) \to \texttt{Prop}$$

$$\models_{\text{Hoare}} \{p\}\ S\ \{q\} \triangleq \forall \mathbf{z}\colon \mathbf{T} \cdot \forall \sigma\colon \Sigma \cdot p(\mathbf{z})(\sigma) \Rightarrow \exists \tau\colon \Sigma \cdot (\sigma \xrightarrow{S} \tau) \land q(\mathbf{z})(\tau)$$

It is straightforward to redefine the relation $\vdash_{\text{Hoare}}$ under this extended interpretation while preserving all of the above results.

Example 4.9. In LEGO, we represent the specification that a program S leaves the value of the program variable x invariant by

$$\vdash_{\text{Hoare}} \{\lambda \mathbf{z}{:}\,\mathbf{T} \cdot \lambda\sigma{:}\,\Sigma \cdot \sigma(x) = \mathbf{z}\}\ \mathbf{call}\ \{\lambda \mathbf{z}{:}\,\mathbf{T} \cdot \lambda\tau{:}\,\Sigma \cdot \tau(x) = \mathbf{z}\}$$

with $\mathbf{T} \triangleq \mathbb{N}$. We will however continue to use the *pretty-printed* notation $\vdash_{\text{Hoare}}$ $\{x = \mathbf{z}\}$ **call** $\{x = \mathbf{z}\}$.

Analysing the failed derivation of $\vdash_{\text{Hoare}} \{x = \mathbf{z}\}$ **call** $\{x = \mathbf{z}\}$, we motivate a new consequence rule to achieve completeness: From an instantiation of (14) i.e.,

$$\{n = x + 1 = \mathbf{z} + 1\}\ \mathbf{call}\ \{x = \mathbf{z}\} \vdash_{\text{Hoare}} \{n = x + 1 = \mathbf{z} + 1\}\ \mathbf{call}\ \{x = \mathbf{z}\} \tag{18}$$

we get stuck having to show

$$\{n = x + 1 = \mathbf{z} + 1\}\ \mathbf{call}\ \{x = \mathbf{z}\} \vdash_{\text{Hoare}} \{n = x + 1 = \mathbf{z}\}\ \mathbf{call}\ \{x + 1 = \mathbf{z}\}\ . \tag{19}$$

It is easy to see that at this stage, no rule is applicable. We require a rule similar to the consequence rule

$$\frac{\Gamma \vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ S\ \{q\}} \qquad \text{provided } \forall\sigma,\tau{:}\,\Sigma \cdot p(\sigma) \Rightarrow p_1(\sigma) \wedge q_1(\tau) \Rightarrow q(\tau)$$

Notice that the side-conditions unnecessarily tie the auxiliary variables of the premiss together with those of the conclusion. In particular, we would have to show $x = \mathbf{z} \Rightarrow x + 1 = \mathbf{z}$.

Taking auxiliary variables seriously, assume that $\mathbf{T}$ and $\mathbf{T}_1$ characterise the auxiliary variables' domain in the conclusion and premiss, respectively. Then, *intuitively*, from the class of assumptions $\forall \mathbf{z}_1{:}\,\mathbf{T}_1 \cdot \Gamma \vdash_{\text{Hoare}} \{p_1(\mathbf{z}_1)\}\ S\ \{q_1(\mathbf{z}_1)\}$ we have to show $\forall \mathbf{z}{:}\,T \cdot \Gamma \vdash_{\text{Hoare}} \{p(\mathbf{z})\}\ S\ \{q(\mathbf{z})\}$. We may relax the side-condition by finding for every instance of $\mathbf{z}{:}\,\mathbf{T}$ an instance $\mathbf{z}_1{:}\,\mathbf{T}_1$ such that $\forall\sigma{:}\,\Sigma, \tau{:}\,\Sigma \cdot (p(\mathbf{z})(\sigma) \Rightarrow p_1(\mathbf{z}_1)(\sigma)) \wedge (q_1(\mathbf{z}_1)(\tau) \Rightarrow q(\mathbf{z})(\tau))$ holds. It is even more effective (see also Sect. 4.3) to choose a witness $\mathbf{z}_1$ relative to the values of variables in the initial and final states σ and τ such that the precondition $p(\mathbf{z})(\sigma)$ holds:

$$\frac{\Gamma \vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\Gamma \vdash_{\text{Hoare}} \{p\}\ S\ \{q\}}$$

$$\text{provided } \forall \mathbf{z}{:}\,\mathbf{T} \cdot \forall\sigma,\tau{:}\,\Sigma \cdot p(\mathbf{z})(\sigma) \Rightarrow \exists \mathbf{z}_1{:}\,\mathbf{T}_1 \cdot p_1(\mathbf{z}_1)(\sigma) \wedge (q_1(\mathbf{z}_1)(\tau) \Rightarrow q(\mathbf{z})(\tau)) \tag{20}$$

Lemma 4.10 (Soundness). *Replacing the consequence rule (13) by (20) preserves soundness.*

Proof. From the premiss $\models_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}$ i.e.,

$$\forall \mathbf{z}_1 : \mathrm{T}_1 \cdot \forall \sigma : \Sigma \cdot p_1(\mathbf{z}_1)(\sigma) \Rightarrow \exists \tau : \Sigma \cdot (\sigma \xrightarrow{S} \tau) \wedge q_1(\mathbf{z}_1)(\tau) \ . \tag{21}$$

and the side condition

$$\forall \mathbf{z} : \mathbf{T} \cdot \forall \sigma, \tau : \Sigma \cdot p(\mathbf{z})(\sigma) \Rightarrow \exists \mathbf{z}_1 : \mathrm{T}_1 \cdot p_1(\mathbf{z}_1)(\sigma) \wedge (q_1(\mathbf{z}_1)(\tau) \Rightarrow q(\mathbf{z})(\tau)) \ , \tag{22}$$

we need to show that $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ holds.

Given an auxiliary variable $\mathbf{z}$ and an initial state σ satisfying $p(\mathbf{z})(\sigma)$, we need to find a final state τ such that $\sigma \xrightarrow{S} \tau$ and $q(\mathbf{z})(\tau)$. Combining $p(\mathbf{z})(\sigma)$ and (22), we can extract a witness $\mathbf{z}'$ such that $p_1(\mathbf{z}')(\sigma)$ holds.

We can now employ (21), yielding the desired final state τ with $\sigma \xrightarrow{S} \tau$ and $q_1(\mathbf{z}')(\tau)$. Having discharged the proof obligation $\sigma \xrightarrow{S} \tau$, we show the remaining $q(\mathbf{z})(\tau)$. In the presence of τ, another refinement by (22) yields an auxiliary variable $\mathbf{z}_1$ satisfying $p_1(\mathbf{z}_1)(\sigma)$ and allows us reducing $q(\mathbf{z})(\tau)$ to $q_1(\mathbf{z}_1)(\tau)$. Appealing again to (21), from $p_1(\mathbf{z}_1)(\sigma)$ we may infer that there is a state η satisfying both $\sigma \xrightarrow{S} \eta$ and $q_1(\mathbf{z}_1)(\eta)$. This completes the proof because the states τ and η must be the same given that the programs considered are deterministic. □

4.2 Completeness

In this section, we show that the new consequence rule leads to a complete verification calculus i.e., the correctness of a program employing Hoare logic is derivable whenever a proof relying on the low-level operational semantics exists. The structure of the proof follows the completeness proof for a more elaborate set of rules in [2]. The central theorem directly relates the descriptive power of operational semantics and Hoare logic:

Theorem 4.11 (Most general formula). *For any program S, in Hoare logic, we can derive $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ where the assertion p characterises the set of states in which S terminates and q characterises the set of all final states i.e.,*

$$\vdash_{\text{Hoare}} \left\{ \lambda \mathbf{z}, \sigma : \Sigma \cdot \sigma \xrightarrow{S} \mathbf{z} \right\}\ S\ \{\lambda \mathbf{z}, \tau : \Sigma \cdot \tau = z\} \ .$$

Proof. By induction on the structure of the program S and a nested induction on the structure of the procedure body S_0 in the case $S \triangleq$ **call**. □

Having treated auxiliary variables in a rigorous manner, it is interesting to observe that the main Theorem 4.11 singles out the set of all states as the domain for auxiliary variables i.e., $\mathbf{T} \triangleq \Sigma$. Intuitively, such a restriction ties the auxiliary variables together with the program variables. Let $\{x_1, \cdots, x_n\}$ be the set of all program variables with corresponding types $\{T_1, \ldots, T_n\}$ occurring in some program S. An assertion of the form $\lambda \mathbf{z}{:}\, \mathbf{T}_1 \times \cdots \times \mathbf{T}_n \cdot \lambda\sigma{:}\,\Sigma \cdot P(\mathbf{z}.1, \ldots, \mathbf{z}.n, \sigma)$ is equivalent to $\lambda \mathbf{z}{:}\,\Sigma \cdot \lambda\sigma{:}\,\Sigma \cdot P(\mathbf{z}(x_1), \ldots, \mathbf{z}(x_n), \sigma)$.

Corollary 4.12. *The verification calculus defined as the least relation* $\vdash_{\text{Hoare}}$ *satisfying (7)–(12), (14) and the new consequence rule (20) is complete.*

It is instructive to study the proof of the Completeness Corollary 4.12 to appreciate the role of the main Theorem 4.11, in particular, why it suffices to consider the set of all states as the domain of auxiliary variables.

Proof. Let S be a program and $p, q{:}\,\mathbf{T} \to \Sigma \to \mathtt{Prop}$ be assertions for an arbitrary domain $\mathbf{T}$. Given $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ i.e.,

$$\forall \mathbf{z}{:}\,\mathbf{T} \cdot \forall\sigma{:}\,\Sigma \cdot p(\mathbf{z})(\sigma) \Rightarrow \exists\tau{:}\,\Sigma \cdot (\sigma \xrightarrow{S} \tau) \wedge q(\mathbf{z})(\tau) \tag{23}$$

we need to show that $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ is derivable. Applying the generalised consequence rule (20) to the main Theorem 4.11 with $p_1(\mathbf{z}_1)(\sigma) \triangleq \sigma \xrightarrow{S} \mathbf{z}_1$ and $q_1(\mathbf{z}_1)(\tau) \triangleq \mathbf{z}_1 = \tau$ yields $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$, provided we can satisfy the side condition

$$\forall \mathbf{z}{:}\, T \cdot \forall\sigma,\tau{:}\,\Sigma \cdot p(\mathbf{z})(\sigma) \Rightarrow \exists \mathbf{z}_1{:}\,\Sigma \cdot (\sigma \xrightarrow{S} \mathbf{z}_1) \wedge (\tau = \mathbf{z}_1 \Rightarrow q(\mathbf{z})(\tau)) \tag{24}$$

Clearly, (23) implies (24). □

4.3 The Rule of Adaptation

Most proposed verification calculi for recursive procedures are known to be unsound or incomplete. Patches to calculi often yield an elaborate set of rules or intricate side-conditions [2, 19]. A common approach has been to retain the consequence rule (13) and adopt further rules to achieve completeness. The rule of adaptation has played a central role in previous work. We show that accounting for known problems in Hoare's rule of adaptation leads to a new rule which turns out to be a simple instantiation of our new consequence rule.

Recall that in order to derive $\vdash_{\text{Hoare}} \{x = \mathbf{z}\}\ \mathbf{call}\ \{x = \mathbf{z}\}$, we need to adapt the auxiliary variable $\mathbf{z}$ if we want to prove (19). In such a situation, from (18), a rule for adapting the auxiliary variable $\mathbf{z}$ leads to

$$\{n = x + 1 = \mathbf{z} + 1\}\ \mathbf{call}\ \{x = \mathbf{z}\} \vdash_{\text{Hoare}} \{p\}\ \mathbf{call}\ \{x + 1 = \mathbf{z}\}$$

where the precondition p should be sufficiently weak to satisfy

$$n = x + 1 = \mathbf{z} \Rightarrow p\ .$$

In general, rules of adaptation are of the form

$$\frac{\Gamma \vdash_{\mathbf{Hoare}} \{p_1\}\ S\ \{q_1\}}{\Gamma \vdash_{\mathbf{Hoare}} \{p\}\ S\ \{q\}}$$

for arbitrary assertions p_1, q_1, q, and particular proposals for the adapted precondition p. Ideally, the rule should be left maximal [7] i.e., the precondition p should be the weakest possible satisfying $\models_{\mathbf{Hoare}} \{p\}\ S\ \{q\}$ in the light of $\models_{\mathbf{Hoare}} \{p_1\}\ s\ \{q_1\}$.

Catering for auxiliary variables at the meta level, Hoare [13] has proposed

$$p \triangleq \lambda\sigma{:}\Sigma \cdot \exists \mathbf{z}_1 \cdot p_1(\sigma) \wedge \forall \tau{:}\Sigma \cdot q_1(\tau) \Rightarrow q(\tau) \tag{25}$$

where $\mathbf{z}_1$ is a list of all (auxiliary) variables free in p_1, q_1, but not in q.

However, while adding Hoare's rule of adaptation leads to a complete verification calculus, Morris [17] and Olderog [19] have shown that, in general, (25) is not the weakest precondition. For total correctness, Morris [17] points out two instructive counter examples:

Example 4.13. Let $p_1 \triangleq q_1 \triangleq \lambda\sigma{:}\Sigma \cdot \mathbf{z} > 0 \wedge \sigma(x) > 0$ and $q \triangleq \lambda\sigma{:}\Sigma \cdot \mathbf{z} \geq 0 \wedge \sigma(x) > 0$. The weakest precondition is then given by $\lambda\sigma{:}\Sigma \cdot \mathbf{z} \geq 0 \wedge \sigma(x) > 0$. However, (25) requires the stronger $\lambda\sigma{:}\Sigma \cdot \mathbf{z} > 0 \wedge \sigma(x) > 0$ (modulo equivalence transformations), because the auxiliary variable $\mathbf{z}$ occurs in both premiss and conclusion.

This problem can be solved by formally treating assertions as relations of auxiliary variables and states i.e.

$$p \triangleq \lambda\mathbf{z}{:}\mathbf{T} \cdot \lambda\sigma{:}\Sigma \cdot \exists \mathbf{z}_1{:}\mathbf{T}_1 \cdot p_1(\mathbf{z}_1)(\sigma) \wedge \forall \tau{:}\Sigma \cdot q_1(\mathbf{z}_1)(\tau) \Rightarrow q(\mathbf{z})(\tau) \ . \tag{26}$$

Morris' second example shows that the choice for the auxiliary variable $\mathbf{z}_1$ may have to depend on the value of variables in the final state:

Example 4.14. Let $p_1 \triangleq \lambda\mathbf{z}{:}\mathbb{N} \cdot \lambda\sigma{:}\Sigma \cdot \mathbf{z} = 0 \vee \mathbf{z} = 1$, $q_1 \triangleq \lambda\mathbf{z}{:}\mathbb{N} \cdot \lambda\sigma{:}\Sigma \cdot \sigma(x) \neq \mathbf{z}$ and $q \triangleq \lambda\mathbf{z}{:}\mathbb{N} \cdot \lambda\sigma{:}\Sigma \cdot \sigma(x) \neq 0 \wedge \sigma(x) \neq 1$. The weakest precondition is then given by **true**. However, both (25) and the weaker (26) are equivalent to **false**.

Relaxing the precondition p so that the witness $\mathbf{z}_1$ can benefit from inspecting the final value of program variables according to τ leads to

$$p \triangleq \lambda\mathbf{z}{:}\mathbf{T} \cdot \lambda\sigma{:}\Sigma \cdot \forall \tau{:}\Sigma \cdot \exists \mathbf{z}_1{:}\mathbf{T}_1 \cdot p_1(\mathbf{z}_1)(\sigma) \wedge (q_1(\mathbf{z}_1)(\tau) \Rightarrow q(\mathbf{z})(\tau)) \ . \tag{27}$$

Notice that our rule of adaptation where the precondition p is the weakest possible (27) is a straightforward instantiation of the new consequence rule (20). Conversely, (20) is admissible in the presence of (13) and our rule of adaptation.

4.4 VDM and Recursive Procedures

The decomposition rules of the Vienna Development Method (VDM) [15] are similar in spirit to Hoare logic. A major conceptual contribution of VDM is that it formally captures the fact that, *in practice*, specifications relate the output to the input i.e., the postcondition may refer to both the initial and final state.

Definition 4.15 (Semantics of VDM's decomposition rules). Following Gordon [9], we can represent the meaning of VDM specifications by the relation

$$\models_{\mathrm{VDM}} \{\cdot\} \cdot \{\cdot\} \; : \; (\Sigma \to \mathsf{Prop}) \times \mathsf{prog} \times (\Sigma \to \Sigma \to \mathsf{Prop}) \to \mathsf{Prop}$$
$$\models_{\mathrm{VDM}} \{p\}\; S\; \{q\} \triangleq \forall \sigma{:}\Sigma \cdot p(\sigma) \Rightarrow \exists \tau{:}\Sigma \cdot (\sigma \xrightarrow{S} \tau) \wedge q(\sigma)(\tau)$$

Presentations of VDM are usually restricted to simple imperative programs with local variables. There is a one-to-one correspondence between the rules of Hoare-style $\vdash_{\mathrm{Hoare}}$ and VDM's decomposition rules $\vdash_{\mathrm{VDM}}$. From our rigorous treatment of auxiliary variables for Hoare logic, it is easy to see that specifications in VDM correspond to a particular class of specification in Hoare logic, in which the auxiliary variables are devoted to freezing the values of all program variables prior to execution. More precisely, given an arbitrary precondition $p{:}\,\Sigma \to \mathsf{Prop}$, program S and postcondition $q{:}\,\Sigma \to \Sigma \to \mathsf{Prop}$, the VDM specification $\vdash_{\mathrm{VDM}} \{p\}\; S\; \{q\}$ is derivable if and only if the specification $\vdash_{\mathrm{Hoare}} \{\lambda \mathbf{z}, \sigma{:}\Sigma \cdot \mathbf{z} = \sigma \wedge p(\sigma)\}\; S\; \{q\}$ is derivable in Hoare logic. Guided by this intuition, we can simplify the consequence rule (20) for a scenario where auxiliary variables capture the initial state:

$$\frac{\Gamma \vdash_{\mathrm{VDM}} \{p_1\}\; S\; \{q_1\}}{\Gamma \vdash_{\mathrm{VDM}} \{p\}\; S\; \{q\}}$$
$$\text{provided } \forall \sigma, \tau{:}\Sigma \cdot p(\sigma) \Rightarrow (p_1(\sigma) \wedge (q_1(\sigma)(\tau) \Rightarrow q(\sigma)(\tau)))\ .$$

An equivalent consequence rule for VDM has been proposed by Aczel [1].

We were able to show that this rule plays a similar role in VDM to our new consequence rule in Hoare logic. More precisely, in LEGO, we have shown that simply adding Sokołowski's procedure call rule to the standard presentation of VDM (neglecting local variables) leads already to a sound and complete system .

The success of VDM reinforces that, in the context of Hoare logic, auxiliary variables deserve a rigorous treatment. Furthermore, VDM's approach suggests that in practice it is feasible to confine the domain of auxiliary variables to the state space Σ.

5 Summary

We have formalised Hoare logic and VDM's decomposition rules for imperative programs dealing with recursive procedures and proved soundness and (relative) completeness for both systems. This work has been mechanically checked by the interactive computer-aided proof system LEGO. Under its influence, we were forced to simplify current presentations of verification calculi to *formally* establish soundness and completeness. In particular, based on work by Apt and Meertens, we have shown how a rigorous treatment of auxiliary variables leads to a new consequence rule. As a trivial instance, we have gained an improved rule of adaptation. We have also been able to show that VDM can easily be extended to cope with recursive procedures. This paper has only dealt with total correctness, but we are confident that similar results for partial correctness can also be established.

The LEGO system has been a valuable tool to achieve our results. It stimulated us to search for crisp calculi and helped us keep track of the correct proof obligations, in particular when dealing with completeness. Given the numerous proposed unsound and incomplete verification calculi, it seems appropriate to further investigate how computer-aided proof systems may contribute to research in program verification.

Acknowledgements

This work has been carried out as part of the Community training project *Co-Development of Imperative Programs and their Correctness Proofs in a Type-Theoretic Environment* funded by the European Commission, programme No ERBFMBICT950199. We have also benefited from the European Commission's Esprit Working Group wg 21900 *TYPES*, the British Council/Deutscher Akademischer Austauschdienst's Academic Research Collaboration Programme *Co-Development of Object-Oriented Programs in* LEGO and some funding from EPSRC.

Paul Jackson provided valuable criticisms and suggestions. Discussions with Martin Hofmann and Zhaohui Luo have greatly influenced the inductive presentation of Hoare logic with contexts. I also like to thank Rod Burstall and Healf Goguen for their helpful comments on earlier drafts of this paper.

References

1. Peter Aczel. A system of proof rules for the correctness of iterative programs – some notational and organisational suggestions. Unpublished, August 1982.
2. Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–162, 1990.
3. Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
4. Krzysztof R. Apt and Lambert G. L. T. Meertens. Completeness with finite systems of intermediate assertions for recursive program schemes. *SIAM Journal on Computing*, 9(4):665–671, November 1980.
5. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.

6. P. Cousot. Methods and logics for proving programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 15, pages 841–993. Elsevier, 1990.
7. Ole-Johan Dahl. *Verifiable Programming.* International Series in Computer Science. Prentice Hall, 1992.
8. R.W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
9. Michael J.C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwhistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving (Banff, Alberta)*, number 15 in Workshops in Computing, pages 387–439. Springer, 1991.
10. David Gries. *The Science of Computer Programming*, chapter 16, pages 193–215. Springer, 1981.
11. D. Harel. *First-order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science.* Springer, 1979.
12. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [14].
13. C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971. Also in [14].
14. C.A.R. Hoare and Cliff B. Jones, editors. *Essays in Computing Science.* International Series in Computer Science. Prentice Hall, 1989.
15. Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, 2 edition, 1990.
16. The Lego World Wide Web page. `http://www.dcs.ed.ac.uk/home/lego`.
17. James H. Morris. Comments on "procedures and parameters". Undated and unpublished.
18. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Proceedings of 16th Conference on Foundations of Software Technology and Theoretical Computer Science (Hyderabad, India, December 18–20, 1996)*, volume 1180 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 1996.
19. Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.
20. Stefan Sokołowski. Total correctness for procedures. In J. Gruska, editor, *Sixth Mathematical Foundations of Computer Science (Tatranská Lomnica)*, volume 53 of *Lecture Notes in Computer Science*, pages 475–483. Springer, 1977.