Hoare Logic for Mutual Recursion and Local Variables

David von Oheimb*

Technische Universität München
http://www.in.tum.de/~oheimb/

Abstract. We present a (the first?) sound and relatively complete Hoare logic for a simple imperative programming language including mutually recursive procedures with call-by-value parameters as well as global and local variables. For such a language we formalize an operational and an axiomatic semantics of partial correctness and prove their equivalence.
Global and local variables, including parameters, are handled in a rather straightforward way allowing for both dynamic and simple static scoping.
For the completeness proof we employ the powerful MGF (Most General Formula) approach, introducing and comparing three variants for dealing with complications arising from mutual recursion.
All this work is done using the theorem prover Isabelle/HOL, which ensures a rigorous treatment of the subject and thus reliable results. The paper gives some new insights in the nature of Hoare logic, in particular motivates a stronger rule of consequence and a new flexible Call rule.

Keywords: axiomatic semantics, Hoare logic, mutual recursion, soundness, relative completeness, local variables, call-by-value parameters, Isabelle/HOL.

1 Introduction

Designing a good Hoare logic for imperative languages with mutually recursive procedures and local variables still is an active area of research. By 'good' we mean a provably sound and (relatively) complete calculus that is as simple as possible and thus easy to apply. There are several complications and pitfalls concerning the status of auxiliary variables, initialization of variables, scoping, parameter passing, and mutual recursion. As we will explain in the sequel, the work presented here provides theoretically interesting and practically useful solutions to these problems, and thus is good in the above sense.

Classical verification systems dealing with these subjects – see [1] for an overview — typically neglect mutual recursion and have turned out to be unsound, as mentioned e.g. by [4] and [5], or incomplete, or at least require several auxiliary rules with awkward syntactic side-conditions. Recent investigations tend to be much more precise, e.g. on the role of auxiliary variables, and even employ mechanical theorem provers to reliably prove soundness and completeness results. Here we emphasize the work of Kleymann[1][10],[5] who suggests a Hoare logic of total correctness and proves it sound and relatively complete with the mechanical theorem prover LEGO.

* research funded by the DFG Project Bali, http://isabelle.in.tum.de/Bali/ .

1 formerly Schreiber.

The work described in the present paper has been conducted in the context of Project BALI formalizing the semantics of Java and proving key properties like type soundness[7] formally within the theorem proving system Isabelle/HOL. Introducing an axiomatic semantics for a large subset of Java, we felt that there were several issues like mutual recursion and parameter passing where we could not resort to already established techniques. It turned out to be very practical and fruitful to perform our investigations in the reduced setting of a simple imperative programming language. In this respect we benefit from the pioneering work of Nipkow[6] that deals with the basic language (without procedures and local variables) within Isabelle/HOL.

One could argue that mutual recursion can be reduced to the already established results on single recursion (e.g. of Kleymann) by program transformation. But this would require non-trivial syntactic manipulations, which would be difficult to handle in a precise proof of soundness and unsuitable for practical program verification. Concerning local variables, the only fully formal treatment we know of, given by Kleymann, is a bit involved, so that one shrinks back from transferring it to procedure parameters. We are not aware of any previous work tackling even either of mutual recursion and procedure parameters whose soundness and (relative) completeness has been mechanically verified.

Just a few words on Isabelle/HOL: This is the instantiation of the generic interactive theorem prover Isabelle[8] with Church's version of Higher-Order Logic. The appearance of formulas on Isabelle/HOL is standard (e.g. '$\Longrightarrow$' is the infix implication symbol associating to the right) except that logical equivalence is expressed with the equality symbol. Predicates are functions with Boolean result, and function application is written in curried style, e.g. $f\,x$. Logical constants are declared by giving their name and type, such as $c :: \tau$. Basic definitions are written $c \equiv t$. Types follow the syntax of ML; type abbreviations are introduced simply as equations. A free datatype is defined by listing its constructors together with their argument types, separated by '|'. Isabelle offers powerful verification tools like natural deduction involving several variants of search, tableaux reasoning, general rewriting, and combinations thereof.

We deliberately let the style of presentation of this paper be influenced by the fully formal treatment caused by using Isabelle/HOL, which should give an impression of its rigor. On the other hand, we abstract from technical details as much as possible in order to present our results in a generic way.

2 The IMP_P Programming Language

Winskel[11] has introduced a simple imperative programming language for educational purposes called IMP. We enriched it with procedures and local variables, calling the result IMP_P. The syntax of its statements ("commands") is

$$\begin{array}{rl} \mathit{com} \;=\; & \texttt{SKIP} \;\mid\; \mathit{com};\, \mathit{com} \;\mid\; \mathit{vname} \texttt{ := } \mathit{aexp} \;\mid\; \texttt{LOCAL } \mathit{loc}\texttt{:=}\mathit{aexp} \texttt{ IN } \mathit{com} \\ \mid & \texttt{IF } \mathit{bexp} \texttt{ THEN } \mathit{com} \texttt{ ELSE } \mathit{com} \;\mid\; \texttt{WHILE } \mathit{bexp} \texttt{ DO } \mathit{com} \\ \mid & \texttt{Call } \mathit{pname} \;\mid\; \mathit{vname} \texttt{ := CALL } \mathit{pname}(\mathit{aexp}) \end{array}$$

where the meanings of most of these constructs (`Call` being just an auxiliary one) is what you expect. The types $aexp = state \rightarrow val$ and $bexp = state \rightarrow bool$ represent arithmetic and Boolean expressions, which we do not further specify since we need only their (black-box) semantics. The type *state* has two components, namely the function spaces $globs = glb \rightarrow val$ and $locals = loc \rightarrow val$ representing the stores for global and local variables. The two kinds of variable names are combined into a free datatype $vname = \mathsf{Glb}\ glb \mid \mathsf{Loc}\ loc$ where Glb and Loc act as tags to distinguish them. The types *glb* and *loc* as well as the type of values *val* are left unspecified. The type of procedure names *pname* is also arbitrary, but is required to be finite, [2] as motivated in §5.

We model the procedure declarations of a given program by a function $\mathsf{body} :: pname \rightarrow com$ mapping procedure names to the corresponding procedure bodies. Our meta-theoretic investigations do not require body to be specified further. For simplicity, each procedure has exactly one parameter, which we model by a generic local variable $\mathtt{Arg} :: \mathsf{loc}$, and a result variable $\mathtt{Res} :: \mathsf{loc}$ (where $\mathtt{Res} \neq \mathtt{Arg}$) whose value is returned on procedure exit. These are merely syntactic restrictions avoiding immaterial but cumbersome details like explicit parameter declarations and return statements.

2.1 Operational Semantics

We define the semantics of IMP_P straightforwardly by an evaluation-style operational ("natural") semantics. The evaluation ("execution") of a statement c is described as a relation $\mathsf{evalc} :: (com \times state \times state)\ set$ between an initial state σ and a final state σ', written $\langle c,\sigma\rangle \longrightarrow \sigma'$. For lack of space and since the other inductive rules defining evalc are standard, we give only the relevant ones here:

$$Local\ \frac{\langle c,\ \sigma_0[a\ \sigma_0/X]\rangle \longrightarrow \sigma_1}{\langle \mathtt{LOCAL}\ X\ \mathtt{:=}\ a\ \mathtt{IN}\ c,\ \sigma_0\rangle \longrightarrow \sigma_1[\sigma_0\langle X\rangle/X]}$$

$$CALL\ \frac{\langle \mathtt{Call}\ pn,\ (\mathsf{setlocs}\ \sigma_0\ \mathsf{newlocs})[a\ \sigma_0/\mathtt{Arg}]\rangle \longrightarrow \sigma_1}{\langle X\mathtt{:=CALL}\ pn(a),\sigma_0\rangle \longrightarrow (\mathsf{setlocs}\ \sigma_1\ (\mathsf{getlocs}\ \sigma_0))[X\mathtt{:=}\sigma_1\langle\mathtt{Res}\rangle]}$$

$$Call\ \frac{\langle \mathsf{body}\ pn,\ \sigma_0\rangle \longrightarrow \sigma_1}{\langle \mathtt{Call}\ pn,\ \sigma_0\rangle \longrightarrow \sigma_1}$$

Note that local variables are initialized immediately when being created. The usual notion of procedure call is split into two parts, which will be very useful for the axiomatic semantics. The `CALL` statement replaces the local variables of the caller by the actual parameter of the called procedure as the only (by virtue of newlocs) local variable — thus implementing trivial static scoping — and restores them (except for assigning the result variable) after return. The `Call` statement is responsible for unfolding the procedure body only, thus implementing recursion. If it is invoked directly rather than via `CALL`, it implements dynamic scoping.

[2] This is not a real restriction but a handy trick that avoids explicit well-formedness constraints implying that in any program there is only a finite number of procedures.

The above definition makes use of a few auxiliary values and functions:

newlocs	$::$ *locals*	
setlocs	$::$ *state* $\rightarrow$ *locals* $\rightarrow$ *state*	
getlocs	$::$ *state* $\rightarrow$ *locals*	shorthand: $\sigma\langle X\rangle \;\equiv$ getlocs $\sigma\ X$
_[_ := _]	$::$ *state* $\rightarrow$ *vname* $\rightarrow$ *val* $\rightarrow$ *state*	shorthand: $\sigma[v/X] \equiv \sigma[\mathsf{Loc}\ X{:=}v]$

Our meta theory does not need define them further as it is independent of their meaning. newlocs is intended to yield the empty set of local variables, setlocs sets the local variables component of the state to a given set of variables, and getlocs returns the local variables of the state. The update function _[_ := _] modifies the state at the given point with a new value, i.e. assigns to a (global or local) variable if it already exists, or otherwise allocates and initializes one.

Properties of the evalc relation, for instance determinism, are typically proved via *rule induction*, i.e. induction on the depth of derivations. In contrast, structural induction (on the syntax of statements) is unsuitable in most cases because rules like *Call* yield structural expansion rather than reduction.

3 Axiomatic Semantics for IMP$_P$

Now that we have introduced the language IMP$_P$, we can describe the core of our work, which is its axiomatic semantics ("Hoare logic").

3.1 Assertions and Hoare Triples

Central to any axiomatic semantics is the notion of *assertions*, which describe properties of the program state before and after executing commands. Semantically speaking, assertions are just predicates on the state. We adopt this abstract view (similarly to our semantic view of expressions) and thus avoid talking explicitly on a syntactic level about terms and substitution and their interpretation. In other words, we do a "shallow embedding" of assertions in our (meta-)logic HOL. Thus, the issue of expressiveness of assertions disappears, and our notion of completeness automatically means completeness (basically) in the sense of Cook[2], i.e. completeness relative to the assumptions that all desired assertions can be expressed syntactically and all valid pure HOL formulas can be proved.

Following Kleymann[5], we give the role of auxiliary variables the attention it deserves. *Auxiliary variables*, also known as "logical" variables (as opposed to program variables), are necessary to relate input and output, in particular to express invariance properties. For example, the proposition that a procedure P does not change the contents of a program variable X is formulated as the Hoare triple $\{X{=}Z\}$ `Call` P $\{X{=}Z\}$, which should mean that whenever X has some value Z before calling P, after return it still has the same value. With this interpretation, Z serves as an auxiliary variable that is implicitly universally quantified. Early works on Hoare logic tended to view Z as a free[3] variable, which gives the desired interpretation only if the triple occurs positively, and otherwise

[3] According to standard conventions, such variables are implicitly universally quantified, i.e. $\Gamma \vdash t\ Z$ is read as $\forall Z.\ \Gamma \vdash t\ Z$. Problems arise if Z occurs also in Γ.

gives incorrect results. Viewing Z as an arbitrary (yet fixed) constant preserves correctness, but this approach suffers from incompleteness: having obtained a procedure specification like $\{X{=}Z\}$ `Call` *Quad* $\{Y{=}Z{*}Z\}$, it is often necessary to exploit (i.e., specialize) it for different instantiations of Z, which is impossible if Z is essentially a constant. The classical way out is sets of substitution and adaptation rules involving intricate side-conditions on variable occurrences. A real solution would be explicit quantification like $\forall Z.\ \{P\ Z\}\ c\ \{Q\ Z\}$, but this changes the structure of Hoare triples and makes them more difficult to handle. Instead we prefer implicit quantification at the level of triple validity, given below, making assertions explicitly dependent not only on the state, but also on auxiliary variables.

Which number of auxiliary variables of which types are required of course depends on the application. So we define the type of assertions with a parameter:

$$\alpha\ \mathit{assn} = \alpha \to \mathit{state} \to \mathit{bool}$$

where α may be instantiated as required. Thus the (pretty-printed) postcondition $\{Y{=}Z{*}Z\}$ mentioned above fully formally reads as $\{\lambda Z\ \sigma.\ \sigma\langle Y\rangle{=}Z{*}Z\}$ where $\alpha = \mathit{int}$. In general it is appropriate (and essential) to let α be the whole *state*, such that all program variables can be monitored when constructing an arbitrary relation between initial and final states.

Built on the type $\alpha\ \mathit{assn}$, we model a *Hoare triple* as the (degenerate) datatype $\alpha\ \mathit{triple} = \{\alpha\ \mathit{assn}\}\ \mathit{com}\ \{\alpha\ \mathit{assn}\}$. It is *valid* wrt. partial correctness, written $\models\{P\}c\{Q\}$, iff $\forall Z\ \sigma.\ P\ Z\ \sigma \Longrightarrow \forall \sigma'.\ \langle c,\sigma\rangle \longrightarrow \sigma' \Longrightarrow Q\ Z\ \sigma'$. Note the universal quantification on the auxiliary variable Z motivated above. This preliminary definition will be refined and extended to judgments with assumptions in §4.1.

3.2 Rules not Dealing with Procedures

The remainder of the current section is dedicated to the question of which Hoare-style rules should be given for the axiomatic semantics of IMP_P. For the moment, simple derivation judgments with single triples, written $\vdash\{P\}c\{Q\}$, suffice to capture everything but recursive procedures. So we take the usual rules, with two exceptions.

$$\mathit{Local}\ \frac{\vdash \{P\}\ c\ \{\lambda Z\ \sigma.\ Q\ Z\ (\sigma[\sigma'\langle X\rangle/X])\}}{\vdash \{\lambda Z\ \sigma.\ \sigma'{=}\sigma \land P\ Z\ (\sigma[a\ \sigma/X])\}\ \mathtt{LOCAL}\ X\mathtt{:=}a\ \mathtt{IN}\ c\ \{Q\}}$$

The *Local* rule adapts the pre- and postconditions reflecting the operational semantics directly. To facilitate this, it remembers the initial state in σ' and extracts the value of X with $\sigma'\langle X\rangle$. (The meta variable σ' could also be put as an auxiliary variable, but this would complicate matters unnecessarily.) As opposed to the rule given in [5], this yields a straightforward handling of local variables. In particular, we do not require explicit mechanisms catering for static scoping because local variables are kept separate from global ones and are reset completely on procedure call (see §3.3 below). Another option, suggested in [1], would be to simply alpha rename X in c, but this would require a syntactic side-condition, namely that the new name does not already occur in P, Q and c, and an unpleasant modification of the program text.

$$conseq\ \frac{\begin{array}{l}\forall Z\,\sigma.\ P\ Z\ \sigma \Longrightarrow \exists P'\ Q'.\ \vdash \{P'\}\ c\ \{Q'\} \wedge \\ \qquad\qquad\qquad \forall \sigma'.\ (\forall Z'.\ P'\ Z'\ \sigma \Longrightarrow Q'\ Z'\ \sigma') \Longrightarrow Q\ Z\ \sigma'\end{array}}{\vdash \{P\}\ c\ \{Q\}}$$

Our *conseq* rule is a strengthened version of the generalized rule of consequence discovered by Kleymann. As motivated in [10], it allows adapting the values of the auxiliary variables as required, due to the universal quantification in their interpretation discussed above. Additionally here, the triple in the premise only needs to be derivable if the precondition P holds, and both new pre- and postconditions may depend on the auxiliary variables and the initial state. This allows not only other common structural rules to be derived (rather than asserted), like

$$\top\ \frac{}{\vdash \{P\}c\{\lambda Z\,\sigma.\ \mathsf{True}\}} \qquad \vee\ \frac{\vdash \{P\}c\{Q\} \quad G\vdash \{P'\}c\{Q'\}}{\vdash \{\lambda Z\,\sigma. P\ Z\ \sigma \vee P'\ Z\ \sigma\}c\{\lambda Z\,\sigma. Q\ Z\ \sigma \vee Q'\ Z\ \sigma\}}$$

but also new structural rules, e.g. one facilitating the use of the *Local* and *CALL* rules:

$$export\ \frac{\forall \sigma'.\ \vdash \{\lambda Z\,\sigma.\ \sigma'{=}\sigma \wedge P\ Z\ \sigma\}c\{Q\}}{\vdash \{P\}c\{Q\}}$$

A typical example is the derivation (modulo predicate-logical steps) for the fact that a local variable does not affect outer local variables with the same name:

$$\begin{array}{r}
\top \\
Local \\
conseq \\
export
\end{array}
\begin{array}{c}
\overline{\forall \sigma'.\ \Gamma\vdash \{\lambda Z\,\sigma.\ \mathsf{True}\}\ c\ \{\lambda Z\,\sigma.\ \sigma'\langle X\rangle{=}(\sigma[\sigma'\langle X\rangle/X])\langle X\rangle\}} \\
\overline{\forall \sigma'.\ \Gamma\vdash \{\lambda Z\,\sigma.\ \sigma'{=}\sigma \wedge \mathsf{True}\}\ \mathtt{LOCAL}\ X\mathtt{:=}a\ \mathtt{IN}\ c\ \{\lambda Z\,\sigma.\ \sigma'\langle X\rangle{=}\sigma\langle X\rangle\}} \\
\overline{\forall \sigma'.\ \Gamma\vdash \{\lambda Z\,\sigma.\ \sigma'{=}\sigma \wedge Z{=}\sigma\langle X\rangle\}\mathtt{LOCAL}\ X\mathtt{:=}a\ \mathtt{IN}\ c\{\lambda Z\,\sigma.\ Z{=}\sigma\langle X\rangle\}} \\
\Gamma\vdash \{\lambda Z\,\sigma.\ Z{=}\sigma\langle X\rangle\}\ \mathtt{LOCAL}\ X\mathtt{:=}a\ \mathtt{IN}\ c\ \{\lambda Z\,\sigma.\ Z{=}\sigma\langle X\rangle\}
\end{array}$$

In a similar way, using some properties of getlocs and $\_[\_\mathtt{:=}\_]$, a version of the *Local* rule corresponding to the classical rule leading to dynamic scope (cf. Rule 17 in [1]) can be derived:

$$\frac{\forall v.\ \Gamma\vdash \{\lambda Z\,\sigma.\ P\ Z\ (\sigma[v/X]) \wedge \sigma\langle X\rangle\ =\ a\ (\sigma[v/X])\}\ c\ \{\lambda Z\,\sigma.\ Q\ Z\ (\sigma[v/X])\}}{\Gamma\vdash \{P\}\ \mathtt{LOCAL}\ X\mathtt{:=}a\ \mathtt{IN}\ c\ \{Q\}}$$

3.3 Simple Procedure Rules

When arriving at procedures, one is faced with the problem that in any practical calculus recursion cannot be handled trivially (i.e. by repeated unfolding). As a first step, we adopt the standard solution of introducing Hoare triples as assumptions of judgments, which enables one to cope with recursive calls of an already unfolded procedure by appealing to a suitable assumption. Revising judgments (currently $\vdash \_ :: \alpha\ triple \to bool$) to $\_ \vdash \_ :: \alpha\ triple\ set \to \alpha\ triple \to bool$, we allow putting triples as assumptions into the contexts of the derivation. In order to reflect this revision, we have to add a context Γ to all judgments in the above rules. Next, we add three rules, the first of them being the well-known *CallN* (‘N’ stands for ‘nested’) rule that makes the specification of the currently unfolded procedure available as an assumption when verifying the procedure body. The second rule enables exploiting assumptions.

$$CallN\ \frac{\{\{P\}\ \mathtt{Call}\ pn\ \{Q\}\}{\cup}\Gamma\vdash \{P\}\ \mathsf{body}\ pn\ \{Q\}}{\Gamma\vdash \{P\}\ \mathtt{Call}\ pn\ \{Q\}} \qquad asm\ \frac{t\in \Gamma}{\Gamma\vdash t}$$

The third rule, $CALL$, is responsible for adapting the local variables, resembling the *Local* rule, though it adapts not only one variable. It resets all local variables and binds the parameter, and in the postcondition restores them (remembering the initial state in σ') except for the one receiving the result:

$$\frac{\Gamma \vdash \{P\}\ \texttt{Call}\ pn\ \{\lambda Z\,\sigma.\ Q\ Z\ ((\mathsf{setlocs}\ \sigma\ (\mathsf{getlocs}\ \sigma'))[X\texttt{:=}\sigma\langle\texttt{Res}\rangle])\}}{\Gamma \vdash \{\lambda Z\,\sigma.\ \sigma'{=}\sigma \wedge P\ Z\ ((\mathsf{setlocs}\ \sigma\ \mathsf{newlocs})[a\ \sigma/\texttt{Arg}])\}X\texttt{:=CALL}\ pn(a)\{Q\}}$$

This rule demonstrates how easy it is to include (call-by-value) procedure parameters, which have been left out by [5]. It is inspired by a similar rule from [9], but differs in that it does not have to impose any syntactic restrictions on the variables occuring in the pre- and postconditions.

3.4 Extended Procedure Rules

As we will show in §5.2, the calculus as given up to here is already complete. Yet when using it to verify mutually recursive procedures with non-linear invocation structure, it becomes tedious: since the assumptions about recursive invocations can only be collected stepwise, often large parts of the proof have to be repeated for different invocation contexts. Consider the example of three procedures P, Q and R, where P calls Q and R, Q calls R, and R calls P and Q. Verifying them with the $CallN$ rule yields the following, roughly abstracted, proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{P,Q,R\}\vdash\texttt{Call}\ P}\quad\dfrac{}{\{P,Q,R\}\vdash\texttt{Call}\ Q}}{\vdots\ \{P,Q,R\}\vdash(\text{body of }R)\ \vdots}}{\{P,Q\}\vdash\texttt{Call}\ R}}{\vdots\ \{P,Q\}\vdash(\text{body of }Q)\ \vdots}}{\{P\}\vdash\texttt{Call}\ Q}\quad\dfrac{\dfrac{\dfrac{}{\{P,R\}\vdash\texttt{Call}\ P}\quad\dfrac{\dfrac{\dfrac{}{\{P,Q,R\}\vdash\texttt{Call}\ R}}{\vdots\ \{P,Q,R\}\vdash(\text{body of }Q)\ \vdots}}{\{P,R\}\vdash\texttt{Call}\ Q}}{\vdots\ \{P,R\}\vdash(\text{body of }R)\ \vdots}}{\{P\}\vdash\texttt{Call}\ R}}{\vdots\ \{P\}\vdash(\text{body of }P)\ \vdots}}{\emptyset\vdash\texttt{Call}\ P}$$

The bodies of Q and R each are verified twice, which may be very redundant. This can be avoided by conducting a *simultaneous* rather than nested verification of all procedures involved. Verification condition generators such as [4] take this idea to the extreme by verifying all procedures contained in a program simultaneously, forcing the user to identify in advance a single specification for each procedure suitable to cover all invocation contexts. Our solution — given next — is more flexible because it permits, each time a call to a cluster of mutually recursive procedures is encountered, to verify simultaneously as many procedures as required (but not more) and to identify the necessary specifications locally.

We extend the judgments further to $\_ \Vdash \_ :: \alpha\ \mathit{triple\ set} \rightarrow \alpha\ \mathit{triple\ set} \rightarrow \mathit{bool}$ ($\Gamma \vdash t$ now becomes an abbreviation of $\Gamma \Vdash \{t\}$) and replace the $CallN$ rule by

$$Call\ \frac{\Gamma \cup \{\{P_i\}\texttt{Call}\ i\{Q_i\} \mid i \in ps\} \Vdash \{\{P_i\}\mathsf{body}\ i\{Q_i\} \mid i \in ps\} \qquad p \in ps}{\Gamma \vdash \{P_p\}\ \texttt{Call}\ p\ \{Q_p\}}$$

When using this rule to verify a call of p, one can decide to verify simultaneously an arbitrary family of procedures where ps is the set of their names including p.

Of course, we now need introduction rules for (finite) conjunctions of triples, whereas elimination rules like *subset* may be derived from the others.

$$\mathit{empty}\ \frac{}{\Gamma \Vdash \emptyset} \qquad \mathit{insert}\ \frac{\Gamma \vdash t \quad \Gamma \Vdash ts}{\Gamma \Vdash \{t\} \cup ts} \qquad \mathit{subset}\ \frac{\Gamma \Vdash ts' \quad ts \subseteq ts'}{\Gamma \Vdash ts}$$

Exploiting the simultaneous *Call* rule, the proof tree of the above example collapses to

$$\dfrac{\dfrac{\overline{\{P,Q,R\} \vdash \texttt{Call}\ P} \quad \overline{\{P,Q,R\} \vdash \texttt{Call}\ Q} \quad \overline{\{P,Q,R\} \vdash \texttt{Call}\ R}}{\vdots\ \{P,Q,R\} \vdash (\text{bodies of } P,\ Q \text{ and } R)\ \vdots}}{\emptyset \vdash \texttt{Call}\ P}$$

where no redundancy concerning procedure bodies remains.

Though it is — strictly speaking — not necessary, we found the *cut* rule very useful in applications, as it helps to adapt the premises of judgments. A similar rule, complementing the *subset* rule, is the well-known *weaken* rule. It can be derived from all others by rule induction, or obtained as an immediate consequence of *cut* and a strengthened version of *asm*.

$$\mathit{cut}\ \frac{\Gamma' \Vdash ts \quad \Gamma \Vdash \Gamma'}{\Gamma \Vdash ts} \qquad \mathit{weaken}\ \frac{\Gamma' \Vdash ts \quad \Gamma' \subseteq \Gamma}{\Gamma \Vdash ts} \qquad \mathit{asm}'\ \frac{ts \subseteq \Gamma}{\Gamma \Vdash ts}$$

4 The Proof of Soundness

This section motivates our actual definition of validity for Hoare triples, which is influenced by the proof of soundness outlined thereafter.

4.1 Validity

Validity involving assumptions, $\_ \models\!\!\models \_ :: \alpha\ \mathit{triple\ set} \to \alpha\ \mathit{triple\ set} \to \mathit{bool}$, could be defined as $\Gamma \models\!\!\models ts \equiv (\forall t \in \Gamma.\ \models t) \Longrightarrow (\forall t \in ts.\ \models t)$. This would be reasonable, but when attempting to prove the *Call* rule which adds assumptions about recursive procedure calls, an inductive argument on the depth of these calls is needed. This could be achieved by syntactic manipulations that unfold procedure calls up to a given depth n, as done in [3]. We prefer a semantic approach instead, which is influenced by [9] and [5]. We define a variant of the operational semantics that includes a counter for the *recursive depth* of evaluations, represented by the judgment $\langle \_,\_ \rangle -\_\!\to \_ :: \mathit{com} \to \mathit{state} \to \mathit{nat} \to \mathit{state} \to \mathit{bool}$. The inductive rules using this new form are exactly the same as in §2.1, except for replacing $\longrightarrow$ by $-n\!\to$ and replacing the *Call* rule by

$$\mathit{Call}\ \frac{\langle \mathsf{body}\ pn,\ \sigma_0 \rangle -n\!\to \sigma_1}{\langle \mathtt{CALL}\ pn,\ \sigma_0 \rangle -n{+}1\!\to \sigma_1}$$

This refinement does not affect the semantics, i.e. the parameter n is a mere annotation, stating that evaluation needs to be done only up to recursive depth n. The equivalence $(\langle c,\sigma \rangle \longrightarrow \sigma') = (\exists n.\ \langle c,\sigma \rangle -n\!\to \sigma')$ can be shown by rule induction for each direction, where the '$\Longrightarrow$' direction requires the lemma $\langle c_1,\sigma_1 \rangle -n_1\!\to \sigma_1' \wedge \langle c_2,\sigma_2 \rangle -n_2\!\to \sigma_2' \Longrightarrow \exists n.\ \langle c_1,\sigma_1 \rangle -n\!\to \sigma_1' \wedge \langle c_2,\sigma_2 \rangle -n\!\to \sigma_2'$ which in turn requires non-strictness: $\langle c,\sigma \rangle -n\!\to \sigma' \wedge n \le m \Longrightarrow \langle c,\sigma \rangle -m\!\to \sigma'$.

According to the refined notion of statement execution, the notion of validity for single Hoare triples receives the recursive depth as an extra parameter:

$$\models n{:}\{P\}c\{Q\} \equiv \forall Z\,\sigma.\ P\ Z\ \sigma \Longrightarrow \forall \sigma'.\ \langle c,\sigma\rangle{-}n{\rightarrow}\ \sigma' \Longrightarrow\ Q\ Z\ \sigma'$$

This definition carries over to sets of triples by $|\!\models n{:}ts \equiv \forall t \in ts.\ \models n{:}t$.
Now we can define the final notion of validity including assumptions as

$$\Gamma\,|\!\models ts \equiv \forall n.\ |\!\models n{:}\Gamma \Longrightarrow\ |\!\models n{:}ts$$

This version is strong and detailed enough to perform induction on the recursive depth. On the other hand, when the set of assumptions is empty, it is equivalent to the version given above because the chain $(\emptyset\,|\!\models ts) = (\forall n.\ |\!\models n{:}\emptyset \Longrightarrow\ |\!\models n{:}ts) = (\forall n.\ |\!\models n{:}ts) = (\forall t \in ts.\models t) = ((\forall t \in \emptyset.\models t) \Longrightarrow (\forall t \in ts.\models t))$ holds.

4.2 Actual Soundness Proof

With our new definition of validity we can express soundness as $\emptyset \vdash t \Longrightarrow \emptyset\models t$. This is a direct instance of $\Gamma\,|\!\vdash ts \Longrightarrow \Gamma\,|\!\models ts$, which can be shown by rule induction on the derivation of the Hoare judgments and an auxiliary rule induction for the *Loop* rule. The *Call* rule is the only difficult case, where we benefit from the proof given in [3] suggesting a lemma that in our case reads as

$\Gamma \cup \{\{P_i\}\ \texttt{Call}\ i\ \{Q_i\} \mid i \in ps\}\ |\!\models\ \{\{P_i\}\ \mathsf{body}\ i\ \{Q_i\} \mid i \in ps\} \Longrightarrow$
$|\!\models n{:}\Gamma \Longrightarrow\ |\!\models n{:}\{\{P_i\}\ \texttt{Call}\ i\ \{Q_i\} \mid i \in ps\}$

Here is the point where the bounded recursive depth comes in, as we conduct the proof by induction on n. Doing this, we exploit the simple facts $\models n{+}1{:}t \Longrightarrow \models n{:}t$, $\models 0{:}\{P\}\ \texttt{Call}\ i\ \{Q\}$, and $(\models n{+}1{:}\{P\}\ \texttt{Call}\ i\ \{Q\}) = (\models n{:}\{P\}\ \mathsf{body}\ i\{Q\})$. The *CallN* rule can of course be derived directly from the *Call* rule.

As we can conclude from this section, the only interesting aspect of the proof of soundness is to find a suitable notion of validity capable of capturing an inductive argument on the recursive depth of procedure calls. Of course, due to the number of rules in the operational and axiomatic semantics, in the inductive proofs there are a lot of cases involving some amount of detail to be considered, for which the mechanical theorem prover is of great help.

5 Three Proofs of Completeness

Much more challenging than the proof of soundness is the proof of completeness. Here we benefit heavily from the MGF approach promoted by [5] and others. We extend this approach, which was given for only a single recursive procedure, to several mutually recursive procedures. When dealing with mutual recursion some complications arise, which we overcome in three different ways, each with specific advantages and drawbacks. For lack of space we can describe only proof outlines and mention crucial lemmas.

5.1 The MGF Approach

For proving completeness of Hoare logics involving procedures, typically some variant of *Most General Formula*, *MGF* for short, is used. A MGF is a judgment $\Gamma \vdash \texttt{MGT}\ c$ where MGT takes a command c and returns a *Most General Triple* which describes the most general property of c, namely its operational semantics. The basic variant of a MGT for partial correctness is

$$\texttt{MGT}\ c \equiv \{\lambda Z\ \sigma_0.\ Z = \sigma_0\}\ c\ \{\lambda Z\ \sigma_1.\ \langle c,Z\rangle \longrightarrow \sigma_1\}$$

Its precondition stores the initial state σ_0 in the auxiliary variable Z, which is consequently of type *state* here. Its postcondition claims that if the execution of command c terminates in some state σ_1, this is the same as the outcome of the operational semantics of c, starting also from σ_0.

Common to all variants of MGTs is that once the corresponding MGF has been proved, completeness almost immediately emerges by virtue of the rule of consequence. For instance, $\emptyset \vdash \texttt{MGT}\ c \Longrightarrow \emptyset \models \{P\}c\{Q\} \Longrightarrow \emptyset \vdash \{P\}c\{Q\}$ can be proved in a two-line Isabelle script applying the definition of validity.

5.2 Version 1: Nested Structural Induction

The outline, proposed by Martin Hofmann[3], of our first completeness proof employs two inductions (in very similar situations) on the structure of commands and a variant of MGT that is a bit more involved, namely

$$\texttt{MGT'}\ c \equiv \{\lambda Z\ \sigma_0.\ \forall \sigma_1.\ \langle c,\sigma_0\rangle \longrightarrow \sigma_1 \Longrightarrow Z = \sigma_1\}\ c\ \{\lambda Z\ \sigma_1.\ Z = \sigma_1\}$$

We refine the outline a little, first by factoring out structural induction into the *MGT-lemma* $(\forall p.\ \Gamma \vdash \texttt{MGT}\ (\texttt{Call}\ p)) \Longrightarrow \Gamma \vdash \texttt{MGT}\ c$ such that it is performed only once, and second by replacing MGT' by the simpler MGT. [4]

The proof of $\emptyset \vdash \texttt{MGT}\ c$ reveals the crux of structural induction: when arriving at unfolding procedure calls, the new subgoal gets structurally larger, such that we cannot appeal directly to any induction hypothesis. Assumptions in the judgments come to the rescue. Still, there remains a challenge: when using them naively, one is faced with the need to use structural induction nested as deep as the number of procedures in the program. This problem is overcome by resorting to an auxiliary induction on the number of procedures not yet considered, such that we strengthen our proof goal to $\Gamma' = \{\texttt{MGT}\ (\texttt{Call}\ p) \mid \mathsf{True}\} \Longrightarrow$ $\forall \Gamma.\ \Gamma \subseteq \Gamma' \Longrightarrow n \le |\Gamma'| \Longrightarrow |\Gamma| = |\Gamma'| - n \Longrightarrow \forall c.\ \Gamma \vdash \texttt{MGT}\ c$ where Γ' equals the set of all possible procedure calls. Its proof is by induction on n, exploiting the MGT-lemma twice. It heavily depends on Γ' being finite as otherwise calculations on cardinality like $|\Gamma| = |\Gamma'| - n$ would be meaningless. Now, $\emptyset \vdash \texttt{MGT}\ c$ is an immediate consequence (just specialize Γ to $\emptyset$ and n to $|\Gamma'|$).

[4] For the case of the WHILE loop, we return to MGT' because there the auxiliary variable σ_0 has to serve as the (invariant) final state of the iteration. Both variants are equivalent, where MGT' entails MGT only if the language is deterministic (which is true for IMP_P) and there are at least two different program states, which we simply assume since empty or singleton state spaces are of no interest anyway.

Note that this version of completeness proof gets by with the *CallN* version of the *Call* rule (thus not requiring the rules *empty* and *insert*), but on the other hand needs to apply it in a nested way.

5.3 Version 2: Simultaneous Structural Induction

Our desire to circumvent the nesting problem of Version 1 has been the motivation for inventing the *Call* rule as an extension of *CallN*, which allows handling procedures simultaneously. Version 2 is also by structural induction and makes use of the MGT-lemma, but by exploiting the power of the *Call* rule, it takes only a much simpler lemma, namely $F \subseteq \{\texttt{MGT}\ (\mathsf{body}\ p) \mid \mathsf{True}\} \Longrightarrow$ $\{\texttt{MGT}\ (\texttt{Call}\ p) \mid \mathsf{True}\} \Vdash F$. The latter is proved by induction on the size of F, so finiteness is vital also here. Comparing Version 2 with Version 1, it requires a more advanced *Call* rule (and the two simple structural rules *empty* and *insert*), but handles mutual recursion more directly and thus clearly.

5.4 Version 3: Rule Induction

Our third version of completeness proof takes the MGF approach to the extreme. It gave us surprising insights into the nature of Hoare logic, yet is probably of mainly theoretic interest because we could not avoid supporting it with two additional rules. Our intuition when discovering this approach has been that structural induction is not too nice, in particular when handling recursion, as the other versions show. Let us employ a more direct and powerful induction scheme: rule induction on the operational semantics.

The pattern of rule induction requires that the inductively defined relation, evalc here, occurs negatively in the formula to be proved. Unfortunately, neither $\emptyset \models \{P\}\ c\ \{Q\} \Longrightarrow \emptyset \vdash \{P\}\ c\ \{Q\}$ itself nor $\emptyset \vdash \texttt{MGT}\ c$ are of this pattern. Let us resort to $\forall \sigma_0\ \sigma_1.\ \langle c,\sigma_0\rangle \longrightarrow \sigma_1 \Longrightarrow \emptyset \vdash \{\lambda Z\,\sigma.\ \sigma = \sigma_0\} c \{\lambda Z\,\sigma.\ \sigma = \sigma_1\}$ which is a kind of MGF property where the evalc relation has been pulled out of the assertion into the meta logic. From this formula we can easily show completeness applying our strong rule of consequence, but we have to require the (clearly admissible, yet non-derivable) extra rule

$$\mathit{diverg}\ \frac{}{G \vdash \{\lambda Z\,\sigma.\ \neg\exists\sigma'.\ \langle c,\, \sigma\rangle \longrightarrow \sigma'\} c \{Q\}}$$

The above MGF property itself is directly amenable to the desired rule induction, which yields a surprisingly short proof. Unfortunately, it requires an unfolding variant of the *Loop* rule reflecting the operational semantics:

$$\mathit{LoopT}\ \frac{\Gamma \vdash \{\lambda Z\,\sigma.\ P\ Z\,\sigma \wedge b\ \sigma\} c \{Q\} \quad \Gamma \vdash \{Q\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{R\}}{\Gamma \vdash \{\lambda Z\,\sigma.\ P\ Z\,\sigma \wedge b\ \sigma\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{R\}}$$

On the other hand, only a trivial variant of the *Call* rules (namely one without assumptions) and no auxiliary variables are needed here.

Thus we can conclude that, in principle, the issues of assumptions and auxiliary variables can be circumvented! Of course, this is only a theoretical point as in actual program verification one does not want to be faced with the operational semantics again, which was suitable for the meta-level completeness proof only.

6 Conclusion

In this paper we have described new approaches for dealing with mutual recursion, procedure parameters and local variables in a Hoare-style calculus. The calculus is powerful — and also simple and convenient — enough to be used in actual program verification efforts. In particular, we have introduced a relatively simple handling of local variables, a convenient and flexible rule for simultaneously verifying mutually recursive procedures, and a strong rule of consequence.

All results have been achieved using the theorem prover Isabelle/HOL, which not only gives full confidence in their correctness, but also was a great aid in cleanly formalizing the theory and conveniently conducting the proofs.

We have combined several existing techniques with new ideas, resulting in a lucid soundness proof and three variants of completeness proofs. Once discovered, they should be transferable to other logical systems and programming languages with relative ease. The major current application is to an object-oriented language, namely the investigation of Java within Project BALI.

Acknowledgments I thank Tobias Nipkow and Martin Hofmann for fruitful discussions on handling mutual recursion. The idea how to perform nested structural induction is due to Martin Hofmann. I also thank Manfred Broy, Tobias Nipkow, Leonor Prensa Nieto, Bernhard Reus, Francis Tang, Markus Wenzel and several anonymous referees for their comments on draft versions of this paper.

References

1. K. R. Apt. Ten years of Hoare logic: A survey — part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.
2. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.
3. Martin Hofmann. Semantik und Verifikation. Lecture notes, in German. http://www.dcs.ed.ac.uk/home/mxh/teaching/marburg.ps.gz, 1997.
4. Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of CADE-13*, volume 1104 of *LNAI*, pages 201–215. Springer-Verlag, 1996.
5. Thomas Kleymann. Hoare logic and VDM: Machine-checked soundness and completeness proofs. (Phd Thesis), ECS-LFCS-98-392, LFCS, 1998.
6. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *FST&TCS*, volume 1180 of *LNCS*, pages 180–192. Springer-Verlag, 1996.
7. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1999.
8. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. Up-to-date description: http://isabelle.in.tum.de/.
9. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.
10. Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 697–711. Springer-Verlag, 1997.
11. Glynn Winskel. *Formal Semantics of Programming Languages*. MIT Press, 1993.