HOARE LOGICS IN ISABELLE/HOL

TOBIAS NIPKOW
Technische Universität München, Institut für Informatik

Abstract. This paper describes Hoare logics for a number of imperative language constructs, from while-loops via exceptions to mutually recursive procedures. Both partial and total correctness are treated. In particular a proof system for total correctness of recursive procedures in the presence of unbounded nondeterminism is presented. All systems are formalized and shown to be sound and complete in the theorem prover Isabelle/HOL.

1. Introduction

Hoare logic is a well developed area with a long history (by computer science standards). The purpose of this report is

- to present, in a unified notation, Hoare logics for a number of different programming language constructs such as loops, exceptions, expressions with side effects, and procedures, together with clear presentations of their soundness and especially completeness proofs, and
- to show that this can and argue that this should be done in a theorem prover, in our case Isabelle/HOL [20].

Thus one can view this report as a relative of Apt's survey papers [2, 3], but with new foundations. Instead of on paper, all formalizations and proofs have been carried out with the help of a theorem prover. A first attempt in this direction [16, 17] merely formalized and debugged an existing completeness proof. Kleymann [27] went one step further and formalized a new and slick Hoare logic for total correctness of recursive procedures. This problem has an interesting history. The logic presented by Apt [2] was later found to be unsound by America and de Boer [1], who modified the system and proved its soundness and completeness. Their proof system, however, suffers from three additional rules with syntactic side conditions. The first really simple system is the one by Kleymann [27] — and it was embedded in a theorem prover.

It may be argued that Kleymann's proof system has nothing to do with the use of a theorem prover. Although the two things are indeed independent of each other, theorem provers tend to act like Occam's razor: the enormous amount of detail one has to deal with when building a model of anything inside a theorem prover often forces one to discover unexpected simplifications. Thus programming logics are an ideal application area for theorem provers: both correctness and simplicity of such logics is of vital importance, and, as Apt himself says [2], "various proofs given in the literature are awkward, incomplete, or even incorrect."

In modelling the assertion language, we follow the *extensional* approach [15] where assertions are identified with functions from states to propositions in the logic of the theorem prover. That is, we model only the semantics but not the syntax of assertions. This is common practice (with the exception of [7], but they do not consider completeness) because the alternative, embedding an assertion language with quantifiers in a theorem prover, is not just hard work but also orthogonal to the problem of embedding the computational part of the Hoare logic. As a consequence we have solved the question of *expressiveness*, i.e. whether the assertion language is strong enough to express all intermediate predicates that may arise in a proof, by going to a higher order logic. Thus our completeness results do not automatically carry over to other logical systems, say first order arithmetic. The advantage of the extensional approach is that it separates reasoning about programs from expressiveness considerations — the latter can then be conducted in isolation for each assertion language.

Much of this report is inspired by the work of Kleymann [10]. He used the theorem prover LEGO [26] rather than Isabelle/HOL, which makes very little difference except in one place (Sect. 2.4). Although in the end, none of our logics are identical to any of his, they are closely related. The main differences are that we consider many more language constructs, we generalize from deterministic to nondeterministic languages, and we consider partial as well as total correctness.

The whole paper is generated directly from the Isabelle input files (which include the text as comments). That is, if you see a lemma or theorem, you can be sure its proof has been checked by Isabelle. But as the report does not go into the details of the proofs, no previous exposure to theorem provers is a prerequisite for reading it.

Isabelle/HOL is an interactive theorem prover for HOL, higher order logic. Most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight some of Isabelle's nonstandard notation.

The syntax $[\![P;\ Q]\!] \Longrightarrow R$ should be read as an inference rule with the two premises P and Q and the conclusion R. Logically it is just a shorthand for $P \Longrightarrow Q \Longrightarrow R$. *Note that semicolon will also denote sequential composition of programs!* There are actually two implications $\longrightarrow$ and $\Longrightarrow$. The two mean the same thing, except that $\longrightarrow$ is HOL's "real" implication, whereas $\Longrightarrow$ comes from Isabelle's meta-logic and expresses inference rules. Thus $\Longrightarrow$ cannot appear inside a HOL formula. For the purpose of this paper the two may be identified. However, beware that $\longrightarrow$ binds more tightly than $\Longrightarrow$: in $\forall x.\ P \longrightarrow Q$ the $\forall x$ covers $P \longrightarrow Q$, whereas in $\forall x.\ P \Longrightarrow Q$ it covers only P.

Set comprehension is written $\{x.\ P\}$ rather than $\{x \mid P\}$ and is also available for tuples, e.g. $\{(x,\ y,\ z).\ P\}$.

1.1. STRUCTURE OF THE PAPER

In Sect. 2 we discuss a simple while-language and modular extensions to nondeterminism and local variables. In Sect. 3 we add exceptions and in Sect. 4 side effects in expression evaluation. In Sect. 5 we add a single parameterless but potentially recursive procedure, later extending it with nondeterminism and local variables. In Sect. 6 the single procedure is replaced by multiple mutually recursive procedures. In Sect. 7 we treat a single function with a single value parameter.

In each case we present syntax, operational semantics and Hoare logic for partial correctness, together with soundness and completeness theorems. For loops and procedures, we also cover total correctness.

The guiding principle is to cover each language feature in isolation and in the simplest possible version. Combinations are only discussed if there is some interference. We do not expect the remaining combinations to present any problems not encountered before.

The choice of language features covered was inspired by the work of von Oheimb [22, 23] who presents an Isabelle/HOL embedding of a Hoare logic for a subset of Java. Based on the proof systems in this report, we have meanwhile designed a simpler Hoare logic for a smaller subset of Java [24].

2. A simple while-language

2.1. SYNTAX AND OPERATIONAL SEMANTICS

We start by declaring the two types *var* and *val* of variables and values:

typedecl *var*
typedecl *val*

They need not be refined any further. Building on them, we define states as functions from variables to values and boolean expressions (*bexp*) as functions from states to the booleans:

types *state* = *var* ⇒ *val*
bexp = *state* ⇒ *bool*

Most of the time the type of states could have been left completely unspecified, just like the types *var* and *val*.

Our model of boolean expressions requires a few words of explanation. Type *bool* is HOL's predefined type of propositions. Thus all the usual logical connectives like ∧ and ∨ are available. Instead of modelling the syntax of boolean expressions, we model their semantics. For example, the programming language expression `x != y` becomes $\lambda s.\ s\ x \neq s\ y$, where $s\ x$ expresses the lookup of the value of variable x in state s.

Now it is time to describe the (abstract and concrete) syntax of our programming language, which is easily done with a recursive datatype such as found in most functional programming languages:

datatype *com* = *Do* (*state* ⇒ *state*)
| *Semi com com* (-; - [60, 60] 10)
| *Cond bexp com com* (*IF* - *THEN* - *ELSE* - 60)
| *While bexp com* (*WHILE* - *DO* - 60)

Statements in this language are called *commands*. They are modelled as terms of type *com*. *Do f* represents an atomic command that changes the state from s to $f\ s$ in one step. Thus the command that does nothing, often called `skip`, can be represented by *Do* ($\lambda s.\ s$). More interestingly, an assignment `x := e`, where `e` is some expression, can be modelled as follows: represent `e` by a function e from *state* to *val*, and the assignment by *Do* ($\lambda s.\ s(x := e\ s)$), where $f(a := v)$ is a predefined construct in HOL for updating function f at argument a with value v. Again we have chosen to model the semantics rather than the syntax, which simplifies matters enormously. Of course it means that we can no longer talk about certain syntactic matters, but that is just fine.

The constructors *Semi*, *Cond* and *While* represent sequential composition, conditional and while-loop. The annotations allow us to write

c1; *c2* *IF b THEN c1 ELSE c2* *WHILE b DO c*

instead of *Semi c1 c2*, *Cond b c1 c2* and *While b c*.

Now it is time to define the semantics of the language, which we do operationally, by the simplest possible scheme, a so-called *evaluation* or *big step* semantics. Execution of commands is defined via triples of the form $s\ -c\rightarrow\ t$ which should be read as "execution of c starting in state

s may terminate in state t". This allows for two kinds of nondeterminism: there may be other executions $s\ -c\rightarrow\ u$ with $t \neq u$, and there may be nonterminating computations as well. For the time being we do not model nontermination explicitly. Only if for some s and c there is no triple $s\ -c\rightarrow\ t$ does this signal what we intuitively view as nontermination. We start with a simple deterministic language and assertions about terminating computations. Nondeterminism and nontermination are treated later. The semantics of our language is defined inductively. Beware that semicolon is used both as a separator of premises and for sequential composition.

$$s\ -Do\ f\rightarrow\ f\ s$$

$$[\![\ s0\ -c1\rightarrow\ s1;\ s1\ -c2\rightarrow\ s2\ ]\!] \Longrightarrow s0\ -c1;c2\rightarrow\ s2$$

$$[\![\ \ b\ s;\ s\ -c1\rightarrow\ t\ ]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2\rightarrow\ t$$
$$[\![\ \neg b\ s;\ s\ -c2\rightarrow\ t\ ]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2\rightarrow\ t$$

$$\neg b\ s \Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow\ s$$
$$[\![\ b\ s;\ s\ -c\rightarrow\ t;\ t\ -WHILE\ b\ DO\ c\rightarrow\ u\ ]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow\ u$$

2.2. HOARE LOGIC FOR PARTIAL CORRECTNESS

We continue our semantic approach by modelling assertions just like boolean expressions, i.e. as functions:

types $assn = state \Rightarrow bool$

Hoare triples are triples of the form $\{P\}\ c\ \{Q\}$, where the assertions P and Q are the so-called pre and postconditions. Such a triple is *valid* (denoted by $\models$) iff every (terminating) execution starting in a state satisfying P ends up in a state satisfying Q:

$$\models \{P\}c\{Q\}\ \equiv\ \forall s\ t.\ s\ -c\rightarrow\ t \longrightarrow P\ s \longrightarrow Q\ t$$

The $\equiv$ sign denotes definitional equality.

This notion of validity is called *partial correctness* because it does not require termination of c.

Finally we come to the core of this paper, Hoare logic, i.e. inference rules for deriving (hopefully valid) Hoare triples. As usual, derivability is indicated by $\vdash$, and defined inductively:

$$\vdash \{\lambda s.\ P(f\ s)\}\ Do\ f\ \{P\}$$

$$[\![\vdash \{P\}c1\{Q\};\ \vdash \{Q\}c2\{R\}\ ]\!] \implies \vdash \{P\}\ c1;c2\ \{R\}$$

$[\![\vdash \{\lambda s.\ P\ s \wedge b\ s\}\ c1\ \{Q\};\ \vdash \{\lambda s.\ P\ s \wedge \neg b\ s\}\ c2\ \{Q\}]\!]$
$\Longrightarrow \vdash \{P\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ \{Q\}$

$\vdash \{\lambda s.\ P\ s \wedge b\ s\}\ c\ \{P\} \implies \vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg b\ s\}$

$[\![\forall s.\ P'\ s \longrightarrow P\ s;\ \vdash \{P\}c\{Q\};\ \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \implies \vdash \{P'\}c\{Q'\}$

The final rule is called the *consequence rule*.

Soundness is proved by induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$:

theorem $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$

Only the *While*-case requires additional help in the form of a lemma:

$[\![s\ -WHILE\ b\ DO\ c\rightarrow\ t;\ P\ s;\ \forall s\ s'.\ P\ s \wedge b\ s \wedge s\ -c\rightarrow\ s' \longrightarrow P\ s']\!]$
$\Longrightarrow P\ t \wedge \neg\ b\ t$

This lemma is the operational counterpart of the *While*-rule of Hoare logic. It is proved by induction on the derivation of $s\ -WHILE\ b\ DO\ c\rightarrow\ t$.

Completeness is not quite as straightforward, but still easy. The proof is best explained in terms of the *weakest precondition*:

$wp :: com \Rightarrow assn \Rightarrow assn$
$wp\ c\ Q \equiv \lambda s.\ \forall t.\ s\ -c\rightarrow\ t \longrightarrow Q\ t$

Loosely speaking, $wp\ c\ Q$ is the set of all start states such that all (terminating) executions of c end up in Q. This is appropriate in the context of partial correctness. Dijkstra calls this the weakest *liberal* precondition to emphasize that it corresponds to partial correctness. We use "weakest precondition" all the time and let the context determine if we talk about partial or total correctness — the latter is introduced further below.

The following lemmas about *wp* are easily derived:

lemma $wp\ (Do\ f)\ Q = (\lambda s.\ Q(f\ s))$
lemma $wp\ (c1;c2)\ R = wp\ c1\ (wp\ c2\ R)$
lemma $wp\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ Q = (\lambda s.\ wp\ (if\ b\ s\ then\ c1\ else\ c2)\ Q\ s)$
lemma $wp\ (WHILE\ b\ DO\ c)\ Q =$
$(\lambda s.\ if\ b\ s\ then\ wp\ (c;WHILE\ b\ DO\ c)\ Q\ s\ else\ Q\ s)$

Note that *if−then−else* is HOL's predefined conditional expression.

By induction on c one can easily prove

lemma $\forall Q. \vdash \{wp\ c\ Q\}\ c\ \{Q\}$

from which completeness follows more or less directly via the rule of consequence:

theorem $\models \{P\}c\{Q\} \implies \vdash \{P\}c\{Q\}$

We discuss two modular extensions of our simple while-language: nondeterminism and local variables. By modularity we mean that we can add these features without disturbing the existing setup. In fact, even the proofs can be extended modularly: the soundness proofs acquire two new easy cases, and for the completeness proofs we merely have to provide suitable lemmas about how *wp* behaves on the new constructs.

2.3.1. *Nondeterminism*

We add a choice construct at the level of commands: $c1 \mid c2$ is the command that can nondeterministically choose to execute either $c1$ or $c2$:

$$s\ -c1\rightarrow\ t \Longrightarrow s\ -c1 \mid c2\rightarrow\ t \qquad s\ -c2\rightarrow\ t \Longrightarrow s\ -c1 \mid c2\rightarrow\ t$$

The proof rule is analogous. If we want to make sure that all executions of $c1 \mid c2$ fulfill their specification, both $c1$ and $c2$ must do so:

$$[\![\vdash \{P\}\ c1\ \{Q\}; \vdash \{P\}\ c2\ \{Q\}]\!] \Longrightarrow \vdash \{P\}\ c1 \mid c2\ \{Q\}$$

The behaviour of *wp* (required for the completness proof) is obvious:

$$wp\ (c1 \mid c2)\ Q = (\lambda s.\ wp\ c1\ Q\ s \wedge wp\ c2\ Q\ s)$$

2.3.2. *Local variables*

We add a new command $VAR\ x = e;\ c$ that assigns x the value of e, executes c, and then restores the old value of x:

$$s(x := e\ s)\ -c\rightarrow\ t \Longrightarrow s\ -VAR\ x = e;\ c\rightarrow\ t(x := s\ x)$$

The corresponding proof rule

$$\forall v. \vdash \{\lambda s.\ P\ (s(x := v)) \wedge s\ x = e\ (s(x := v))\}\ c\ \{\lambda s.\ Q\ (s(x := v))\} \Longrightarrow \vdash \{P\}\ VAR\ x = e;\ c\ \{Q\}$$

needs a few words of explanation. The main problem is how to refer to the initial value of x in the postcondition. In some related calculi like VDM [9], this is part of the logic, but in plain Hoare logic we have to remember the old value of x explicitly by equating it to something we can refer to in the postcondition. That is the *raison d'être* for v. Of course this should work for every value of x. Hence the $\forall v$. If you are used to more syntactic presentations of Hoare logic you may prefer a side condition that v does not occur free in P, e, c or Q. However, since we embed Hoare logic in a language with quantifiers, why not use them to good effect?

The behaviour of *wp* mirrors the execution of *VAR*:

$$wp\ (VAR\ x = e;\ c)\ Q = (\lambda s.\ wp\ c\ (\lambda t.\ Q\ (t(x := s\ x)))\ (s(x := e\ s)))$$

2.4. HOARE LOGIC FOR TOTAL CORRECTNESS

2.4.1. *Termination*

Although partial correctness appeals because of its simplicity, in many cases one would like the additional assurance that the command is guaranteed to termiate if started in a state that satisfies the precondition. Even to express this we need to define when a command is guaranteed to terminate. We can do this without modifying our existing semantics by merely adding a second inductively defined judgement $c \downarrow s$ that expresses guaranteed termination of c started in state s:

$$Do\ f \downarrow s$$

$$[\![\ c1 \downarrow s0;\ \forall s1.\ s0\ -c1\rightarrow\ s1 \longrightarrow c2 \downarrow s1\ ]\!] \Longrightarrow (c1;c2) \downarrow s0$$

$$[\![\ b\ s;\ c1 \downarrow s\ ]\!] \Longrightarrow IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s$$
$$[\![\ \neg b\ s;\ c2 \downarrow s\ ]\!] \Longrightarrow IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s$$

$$\neg b\ s \Longrightarrow WHILE\ b\ DO\ c \downarrow s$$
$$[\![\ b\ s;\ c \downarrow s;\ \forall t.\ s\ -c\rightarrow\ t \longrightarrow WHILE\ b\ DO\ c \downarrow t\ ]\!] \Longrightarrow WHILE\ b\ DO\ c \downarrow s$$

The rules should be self-explanatory.

Now that we have termination, we can define total validity, $\models_t$, as partial validity and guaranteed termination:

$$\models_t \{P\}c\{Q\}\ \equiv\ \models \{P\}c\{Q\} \wedge (\forall s.\ P\ s \longrightarrow c{\downarrow}s)$$

2.4.2. *Hoare logic*

Derivability of Hoare triples in the proof system for total correctness is written $\vdash_t \{P\}\ c\ \{Q\}$ and defined inductively. The rules for $\vdash_t$ differ from those for $\vdash$ only in the one place where nontermination can arise: the *While*-rule. Hence we only show that one rule:

$$[\![wf\ r;\ \ \forall s'.\ \vdash_t \{\lambda s.\ P\ s \wedge b\ s \wedge s' = s\}\ c\ \{\lambda s.\ P\ s \wedge (s,s') \in r\}]\!]$$
$$\Longrightarrow \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg b\ s\}$$

The rule is like the one for partial correctness but it requires additionally that with every execution of the loop body a wellfounded relation $(wf\ r)$ on the state space decreases: wellfoundedness of r means there is no infinite

descending chain $\ldots, (s2, s1) \in r, (s1, s0) \in r$. To compare the value of the state before and after the execution of the loop body we again use the trick discussed in connection with local variables above: a locally $\forall$-quantified variable.

This is almost the rule by Kleymann [10], except that we do not have a wellfounded relation on some arbitrary set *together* with a measure function on states, but have collapsed this into a wellfounded relation on states. This does not just shorten the rule but it also simplifies it logically: now we know that wellfounded relations on the state space suffice and we do not need to drag in other types. I should mention that this simplification was forced on me by Isabelle: since Isabelle does not allow local quantification over types, I could not even express Kleymann's rule, which requires just that.

The soundness theorem

theorem $\vdash_t \{P\}c\{Q\} \implies \models_t \{P\}c\{Q\}$

is again proved by induction over c. But in the *While*-case we do not appeal to the same lemma as in the proof for $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \models \{P\}\ c\ \{Q\}$. Instead we perform a local proof by wellfounded induction over the given relation r.

The completeness proof proceeds along the same lines as the one for partial correctness. First we have to strengthen our notion of weakest precondition to take termination into account:

$$wp_t\ c\ Q \equiv \lambda s.\ wp\ c\ Q\ s \wedge c{\downarrow}s$$

The lemmas proved about wp_t are the same as those for wp, except for the *While*-case, which we deal with locally below. The key lemma

lemma $\forall Q.\ \vdash_t \{wp_t\ c\ Q\}\ c\ \{Q\}$

is again proved by induction on c. The *While*-case is interesting because we now have to furnish a suitable wellfounded relation. Of course the execution of the loop body directly yields the required relation, as the following lemma shows. Remember that set comprehension in Isabelle/HOL uses "." rather than "|".

$wf\ \{(t, s).\ WHILE\ b\ DO\ c \downarrow s \wedge b\ s \wedge s\ -c\rightarrow\ t\}$

This lemma follows easily from the lemma that if $WHILE\ b\ DO\ c \downarrow s$ then there is no infinite sequence of executions of the body, which is proved by induction on $WHILE\ b\ DO\ c \downarrow s$.

The actual completeness theorem follows directly, in the same manner as for partial correctness.

theorem $\models_t \{P\}c\{Q\} \Longrightarrow \vdash_t \{P\}c\{Q\}$

2.4.3. *Modular extensions of pure while*

Nondeterministic choice and local variables can be added without disturbing anything. Their proof rules remain exactly the same as in the case of partial correctness. We have carried this out as well; the details are straightforward.

3. Exceptions

We extend our pure while-language with exceptions, a modification that is decidedly non-modular as it changes the state space and the semantics.

3.1. SYNTAX AND SEMANTICS

Our exceptions are very simple: there is only one exception, which we call *error*, and it can be raised and handled. Semantically we treat errors by extending our state space with a new boolean component that indicates if the error has been raised. This new state space *estate* is defined as a record with two components:

record *estate* = *st* :: *state*
err :: *bool*

Record selectors are simply projection functions. Records are constructed as in $(\!|st = s,\ err = False|\!)$ and updated selectively as in $es(\!|err := True|\!)$. We also introduce *ok s* as a shorthand for $\neg err\ s$.

Boolean expressions are now defined as

types *bexp* = *estate* ⇒ *bool*

The syntax of the language with errors is the same as the simple while language, but extended with a construct for handling errors:

datatype *com* = *Do* (*estate* ⇒ *estate*)
| *Semi com com* (-; - [60, 60] 10)
| *Cond bexp com com* (*IF - THEN - ELSE - 60*)
| *While bexp com* (*WHILE - DO - 60*)
| *Handle com com* (- *HANDLE - 60*)

How is an error raised? Simply by $Do(\lambda s.\ s(\!|err{:=}True|\!))$. And how is it handled? Command *c1 HANDLE c2* executes *c1*, and, if this raises an error, resets the error and continues with executing *c2*.

Having the error flag as part of the state allows us to execute commands only if the state is *ok* and to skip execution otherwise [19]. It leads to the following set of rules for command execution:

$$ok\ s \Longrightarrow s\ -Do\ f\rightarrow f\ s$$

$$[\![\ s0\ -c1\rightarrow s1;\ s1\ -c2\rightarrow s2\ ]\!] \Longrightarrow s0\ -c1;c2\rightarrow s2$$

$$[\![\ ok\ s;\ \ b\ s;\ s\ -c1\rightarrow t\ ]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2\rightarrow t$$
$$[\![\ ok\ s;\ \neg b\ s;\ s\ -c2\rightarrow t\ ]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2\rightarrow t$$

$$[\![\ ok\ s;\ \neg b\ s\ ]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow s$$
$$[\![\ ok\ s;\ b\ s;\ s\ -c\rightarrow t;\ t\ -WHILE\ b\ DO\ c\rightarrow u\ ]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow u$$

$$[\![\ ok\ s0;\ s0\ -c1\rightarrow s1;\ ok\ s1\ ]\!] \Longrightarrow s0\ -c1\ HANDLE\ c2\rightarrow s1$$
$$[\![\ ok\ s0;\ s0\ -c1\rightarrow s1;\ err\ s1;\ s1(\!|err:=False|\!)\ -c2\rightarrow s2\ ]\!]$$
$$\Longrightarrow s0\ -c1\ HANDLE\ c2\rightarrow s2$$

$$err\ s \Longrightarrow s\ -c\rightarrow s$$

3.2. HOARE LOGIC

The Hoare logic follows the same lines as the one for the language without exceptions. The main change is that assertions are now functions of *estate*:

types *assn* = *estate* ⇒ *bool*

Hence pre and postconditions can talk about whether an error has been raised or not. Validity $\models \{P\}\ c\ \{Q\}$ is again partial correctness as defined in Sect. 2.2. The proof rules for the logic are the following:

$$\vdash \{\lambda s.\ if\ err\ s\ then\ P\ s\ else\ P(f\ s)\}\ Do\ f\ \{P\}$$

$$[\![\vdash \{P\}c1\{Q\}; \vdash \{Q\}c2\{R\}\ ]\!] \Longrightarrow \vdash \{P\}\ c1;c2\ \{R\}$$

$$[\![\vdash \{\lambda s.\ ok\ s \wedge P\ s\ \wedge b\ s\}\ c1\ \{Q\};\ \vdash \{\lambda s.\ ok\ s \wedge P\ s \wedge \neg b\ s\}\ c2\ \{Q\}\ ]\!]$$
$$\Longrightarrow \vdash \{\lambda s.\ if\ err\ s\ then\ Q\ s\ else\ P\ s\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ \{Q\}$$

$$\vdash \{\lambda s.\ ok\ s \wedge P\ s \wedge b\ s\}\ c\ \{P\}$$
$$\Longrightarrow \vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge (ok\ s \longrightarrow \neg b\ s)\}$$

$$[\![\vdash \{\lambda s.\ ok\ s \wedge P\ s\}\ c1\ \{\lambda s.\ if\ err\ s\ then\ Q(s(\!|err:=False|\!))\ else\ R\ s\};$$
$$\vdash \{Q\}\ c2\ \{R\}\ ]\!]$$
$$\Longrightarrow \vdash \{\lambda s.\ if\ err\ s\ then\ R\ s\ else\ P\ s\}\ c1\ HANDLE\ c2\ \{R\}$$

$$[\![\forall s.\ P'\ s \longrightarrow P\ s; \vdash \{P\}c\{Q\}; \forall s.\ Q\ s \longrightarrow Q'\ s\ ]\!] \Longrightarrow \vdash \{P'\}c\{Q'\}$$

The conclusions of the rules for *Do*, *Cond* and *Handle* follow the pattern that the precondition has been augmented with a conditional that reduces

to the "normal" precondition P if no error is present but collapses to the postcondition Q otherwise, because in that case the command does nothing. In the *While* rule we had to ensure that invariance of P only needs to be proved for *ok* states, and that after the loop we can only infer the negation of the loop test if we are in an *ok* state — otherwise we may have left the loop by the error exit instead of normally. The most puzzling rule may be the one for *Handle*: why does it always require $\vdash \{Q\}\ c2\ \{R\}$, even if *c1* cannot raise an error? The answer is that we can simply set Q to *False*, in which case $\vdash \{Q\}\ c2\ \{R\}$ is always provable.

Soundness is proved as usual by induction on c, almost exactly as for the basic while-language in Sect. 2.2, just with a few more case distinctions.

theorem $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$

The weakest precondition *wp* is also defined as in Sect. 2.2, but we obtain a different set of derived laws:

lemma $wp\ (Do\ f)\ Q = (\lambda s.\ if\ err\ s\ then\ Q\ s\ else\ Q(f\ s))$
lemma $wp\ (c1;c2)\ R = wp\ c1\ (wp\ c2\ R)$
lemma $wp\ (IF\ b\ THEN\ c1\ ELSE\ c2)\ Q =$
$(\lambda s.\ if\ err\ s\ then\ Q\ s\ else\ wp\ (if\ b\ s\ then\ c1\ else\ c2)\ Q\ s)$
lemma $wp\ (WHILE\ b\ DO\ c)\ Q =$
$(\lambda s.\ if\ err\ s\ then\ Q\ s\ else\ if\ b\ s\ then\ wp\ (c;WHILE\ b\ DO\ c)\ Q\ s\ else\ Q\ s)$
lemma $wp\ (c1\ HANDLE\ c2)\ R =$
$(\lambda s.\ if\ err\ s\ then\ R\ s$
$else\ wp\ c1\ (\lambda t.\ if\ err\ t\ then\ (wp\ c2\ R)(t(\!|err:=False|\!))\ else\ R\ t)\ s)$

As in Sect. 2.2, the key lemma is now proved without much fuss

lemma $\forall Q. \vdash \{wp\ c\ Q\}\ c\ \{Q\}$

and completeness follows directly:

theorem $\models \{P\}c\{Q\} \implies \vdash \{P\}c\{Q\}$

4. Side effects

We consider a language where the evaluation of expressions may have side effects. In practice this occurs because of side effecting operators like ++ in C or because of user-defined functions with side effects. One trivial solution to this problem is to require a program transformation step that eliminates compound expressions. For example, `x := f(g(y))`, where `f` and `g` may have side effects, is transformed into `z := g(y); x := f(z)`. The resulting program is easy to deal with. Our aim is to show that one can reason about compound expressions directly and that the required Hoare logic is quite

straightforward. The essential idea goes back to Kowaltowski [12]: specify the behaviour of expressions by Hoare triples where the postcondition can refer to the value of the expression. This was already formalized by von Oheimb [22]. We improve his rules a little by dropping the unnecessary dependence of the precondition on the expression value.

4.1. SYNTAX AND SEMANTICS

Types *var*, *val*, *state* and *bexp* are defined as in Sect. 2. But now we introduce a separate type of *expressions* because we want to study how expression evaluation interacts with side effecting function calls:

datatype *expr* = *Var var* | *Fun* (*state* ⇒ *val list* ⇒ *val* × *state*) (*expr list*)

An expression can either be a variable or *Fun f es*, the application of a function f to a list of expressions *es*. Function f depends not just on a list of values but also on the state, and it returns not just a value but also a new state. Thus we now have a more syntactic representation of expressions (e.g. compared with *bexp*), but the notion of functions is still a semantic one.

Throughout this section, e always stands for an expression and *es* always for an expression list.

With the arrival of expressions, the syntax of commands changes: the generic *Do* is replaced with a proper assigment command, and *SKIP* is added as a separate command:

datatype *com* = *SKIP*

| *Assign var expr*	(- := - [60, 60] 10)
| *Semi com com*	(-; - [60, 60] 10)
| *Cond bexp com com*	(*IF* - *THEN* - *ELSE* - 60)
| *While bexp com*	(*WHILE* - *DO* - 60)

Now that expression evaluation can have side effects, the semantics of the language is defined by three transition relations:

$s\ -c\rightarrow\ t$ the familiar execution of commands,

$s\ -e\Rightarrow\ (v,t)$ the evaluation of an expression e which produces both a value v and a new state t, and

$s\ =es\Rightarrow\ (vs,t)$ the evaluation of an expression list *es* which produces both a list of values *vs* and a new state t.

Evaluation of expressions and expression lists is defined mutually inductively:

$s\ -Var\ x\Rightarrow\ (s\ x,s)$
$s\ =es\Rightarrow\ (vs,t) \Longrightarrow s\ -Fun\ f\ es\Rightarrow\ f\ t\ vs$

$s\ =[]\Rightarrow\ ([],s)$
$[\![s\ -e\Rightarrow\ (v,t);\ t\ =es\Rightarrow\ (vs,u)]\!] \Longrightarrow s\ =e\#es\Rightarrow\ (v\#vs,u)$

Lists in Isabelle/HOL are built up from them empty list [] by the infix constructor #, where $x\ \#\ xs$ is the list with head x and tail xs.

Command execution is defined as usual. Hence we only show the rules for the two new commands:

$s\ -SKIP\rightarrow\ s$

$s\ -e\Rightarrow\ (v,t)\ \Longrightarrow\ s\ -x:=e\rightarrow\ t(x:=v)$

4.2. HOARE LOGIC

Since expresssion evaluation may change the state, we need to reason about the individual expresssion evaluation steps as well. To reason about evaluation, we need to take the computed values into account, too. Thus there will be two new kinds of Hoare triples: $\{P\}e\{Q'\}$ and $\{P\}es\{Q''\}$, where P depends only on the state but where Q' depends also on the value of e and Q'' also on the value of es. Thus there are three types of assertions:

types *assn* = *state* ⇒ *bool*
vassn = *val* ⇒ *state* ⇒ *bool*
vsassn = *val list* ⇒ *state* ⇒ *bool*

Most of them time we use P, Q and R for all three kinds of assertions.

Validity of the three kinds of Hoare triples is denoted by $\models$, $\models_e$, $\models_{es}$. The definitions need no comments:

$\models\ \ \{P\}c\{Q\}\ \ \equiv\ \forall s\ t.\ s\ -c\rightarrow\ t\ \longrightarrow\ P\ s\ \longrightarrow\ Q\ t$
$\models_e\ \{P\}e\{Q\}\ \ \equiv\ \forall s\ t\ v.\ s\ -e\Rightarrow\ (v,t)\ \longrightarrow\ P\ s\ \longrightarrow\ Q\ v\ t$
$\models_{es}\ \{P\}es\{Q\}\ \equiv\ \forall s\ t\ vs.\ s\ =es\Rightarrow\ (vs,t)\ \longrightarrow\ P\ s\ \longrightarrow\ Q\ vs\ t$

Thus there are also three kinds of judgements: $\vdash\ \{P\}\ c\ \{Q\}$, $\vdash_e\ \{P\}\ e\ \{Q\}$ and $\vdash_{es}\ \{P\}\ es\ \{Q\}$. The latter two are defined by mutual induction:

$\vdash_e\ \{\lambda s.\ Q\ (s\ x)\ s\}\ Var\ x\ \{Q\}$
$\vdash_{es}\ \{P\}\ es\ \{\lambda vs\ s.\ Q\ (fst(f\ s\ vs))\ (snd(f\ s\ vs))\}\ \Longrightarrow\ \vdash_e\ \{P\}\ Fun\ f\ es\ \{Q\}$

$\vdash_{es}\ \{P\ []\}\ []\ \{P\}$
$[\![\ \vdash_e\ \{P\}\ e\ \{Q\};\ \forall v.\ \vdash_{es}\ \{Q\ v\}\ es\ \{\lambda vs.\ R(v\#vs)\}\ ]\!]\ \Longrightarrow\ \vdash_{es}\ \{P\}\ e\#es\ \{R\}$

Functions *fst* and *snd* select the first and second component of a pair, i.e. the value and the state in the above rule.

If you wonder where the rules come from: they are derived from the proofs one would perform in ordinary Hoare logic on the program one obtains by removing nested expressions as indicated at the beginning of this section. You can still recognize the ordinary assigment axiom (first rule) and sequential composition (last rule).

As for the operational semantics, the rules for commands are the same as in the side effect free language, except of course for the two new commands, whose rules are straightforward:

$$\vdash \{P\}\ SKIP\ \{P\}$$

$$\vdash_e \{P\}\ e\ \{\lambda v\ s.\ Q(s(x{:=}v))\} \implies \vdash \{P\}\ x{:=}e\ \{Q\}$$

Soundness of $\vdash_e$ and $\vdash_{es}$ is easily proved by simultaneous induction on e and es:

theorem *ehoare-sound*:
$(\vdash_e \{P\}e\{Q'\} \longrightarrow \models_e \{P\}e\{Q'\}) \ \wedge\ (\vdash_{es} \{P\}es\{Q''\} \longrightarrow \models_{es} \{P\}es\{Q''\})$

Soundness of $\vdash$ is proved as usual, by induction on c. The *Assign*-case is solved by lemma *ehoare-sound* above.

theorem $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$

Completeness is also proved in the standard manner. But since we have three kinds of triples, we also need three weakest preconditions:

$wp :: com \Rightarrow assn \Rightarrow assn$
$wp\ c\ Q \equiv (\lambda s.\ \forall t.\ s\ -c\rightarrow\ t \longrightarrow Q\ t)$
$wpe :: expr \Rightarrow vassn \Rightarrow assn$
$wpe\ e\ Q \equiv (\lambda s.\ \forall v\ t.\ s\ -e\Rightarrow (v,t) \longrightarrow Q\ v\ t)$
$wpes :: expr\ list \Rightarrow vsassn \Rightarrow assn$
$wpes\ es\ Q \equiv (\lambda s.\ \forall vs\ t.\ s\ =es\Rightarrow (vs,t) \longrightarrow Q\ vs\ t)$

Of the laws proved about *wp*, *wpe* and *wpes* we only show the "new" ones:

lemma $wp\ SKIP\ P = P$
lemma $wp\ (x{:=}e)\ Q = wpe\ e\ (\lambda v\ s.\ Q(s(x{:=}v)))$
lemma $wpe\ (Var\ x)\ Q = (\lambda s.\ Q\ (s\ x)\ s)$
lemma $wpe\ (Fun\ f\ es)\ Q = wpes\ es\ (\lambda vs\ s.\ Q\ (fst(f\ s\ vs))\ (snd(f\ s\ vs)))$
lemma $wpes\ []\ Q = Q\ []$
lemma $wpes\ (e\#es)\ Q = wpe\ e\ (\lambda v.\ wpes\ es\ (\lambda vs.\ Q(v\#vs)))$

Our standard lemma for the completeness theorem is first proved for expressions and expression lists by (an easy) simultaneous induction on e and es:

lemma $(\forall Q.\ \vdash_e \{wpe\ e\ Q\}\ e\ \{Q\}) \ \wedge\ (\forall Q.\ \vdash_{es} \{wpes\ es\ Q\}\ es\ \{Q\})$

With the help of this lemma in the *Assign*-case we can prove the key lemma by induction on c; the other cases go through as usual.

lemma $\forall Q.\ \vdash \{wp\ c\ Q\}\ c\ \{Q\}$

The completeness theorem follows directly:

theorem $\models \{P\}c\{Q\} \implies \vdash \{P\}c\{Q\}$

5. Procedures

So far, things were well-understood long before they were modelled in a theorem prover. Procedures, however, are different. In the introduction I have already sketched the history of Hoare logics for recursive procedures. As a motivation of the technical difficulties, consider the following parameterless recursive procedure:

```
proc = if i=0 then skip else i := i-1; CALL; i := i+1
```

A classic example of the subtle problems associated with reasoning about procedures is the proof that `i` is invariant: {`i=N`} `CALL` {`i=N`}. This is done by induction: we assume {`i=N`} `CALL` {`i=N`} and have to prove {`i=N`} `body` {`i=N`}, where `body` is the body of the procedure. The case `i=0` is trivial. Otherwise we have to show {`i=N`}`i:=i-1;CALL;i:=i+1`{`i=N`}, which can be reduced to {`i=N-1`} `CALL` {`i=N-1`}. But how can we deduce{`i=N-1`} `CALL` {`i=N-1`} from the induction hypothesis {`i=N`} `CALL` {`i=N`}? Clearly, we have to instantiate `N` in the induction hypothesis — after all `N` is arbitrary as it does not occur in the program. The problems with procedures are largely due to unsound or incomplete adaption rules. We follow the solution of Morris and Kleymann and adjust the value of auxiliary variables like `N` with the help of the consequence rule. We also follow Kleymann in modelling auxiliary variables as a separate concept (as suggested in [4]). Our main contribution is a generalization from deterministic to nondeterministic languages.

5.1. SYNTAX AND OPERATIONAL SEMANTICS

Types *var*, *val*, *state* and *bexp* are defined as in Sect. 2. We start with a minimal set of commands:

datatype $com = Do\ (state \Rightarrow state)$
$\quad | \ Semi\ \ com\ com \qquad (\text{-; -}\ \ [60, 60]\ 10)$
$\quad | \ Cond\ \ bexp\ com\ com \qquad (IF\ \text{-}\ THEN\ \text{-}\ ELSE\ \text{-}\ \ 60)$
$\quad | \ While\ bexp\ com \qquad (WHILE\ \text{-}\ DO\ \text{-}\ \ 60)$
$\quad | \ CALL$

There is only one parameterless procedure in the program. Hence *CALL* does not even need to mention the procedure name. There is no separate syntax for procedure declarations. Instead we declare a HOL constant

consts *body* :: *com*

that represents the body of the one procedure in the program.

As before, command execution is described by transitions $s\ -c\rightarrow\ t$. The only new rule is the one for *CALL* — it requires no comment:

$$s\ -body\rightarrow\ t \Longrightarrow s\ -CALL\rightarrow\ t$$

This semantics turns out not to be fine-grained enough. The soundness proof for the Hoare logic below proceeds by induction on the call depth during execution. To make this work we define a second semantics $s\ -c-n\rightarrow\ t$ which expresses that the execution uses at most n nested procedure invocations, where n is a natural number. The rules are straightforward: n is just passed around, except for procedure calls, where it is decremented (*Suc n* is $n\ +\ 1$):

$$s\ -Do\ f-n\rightarrow\ f\ s$$

$$[\![\ s0\ -c1-n\rightarrow\ s1;\ s1\ -c2-n\rightarrow\ s2\ ]\!] \Longrightarrow s0\ -c1;c2-n\rightarrow\ s2$$

$$[\![\ \ b\ s;\ s\ -c1-n\rightarrow\ t\ ]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2-n\rightarrow\ t$$
$$[\![\ \neg b\ s;\ s\ -c2-n\rightarrow\ t\ ]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2-n\rightarrow\ t$$

$$\neg b\ s \Longrightarrow s\ -WHILE\ b\ DO\ c-n\rightarrow\ s$$
$$[\![b\ s;\ s\ -c-n\rightarrow\ t;\ t\ -WHILE\ b\ DO\ c-n\rightarrow\ u]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c-n\rightarrow\ u$$

$$s\ -body-n\rightarrow\ t \Longrightarrow s\ -CALL-Suc\ n\rightarrow\ t$$

By induction on $s\ -c-m\rightarrow\ t$ we show monotonicity w.r.t. the call depth:

lemma $s\ -c-m\rightarrow\ t \Longrightarrow \forall\, n.\ m \leq n \longrightarrow s\ -c-n\rightarrow\ t$

With the help of this lemma we prove the expected relationship between the two semantics:

lemma *exec-iff-execn*: $(s\ -c\rightarrow\ t) = (\exists\, n.\ s\ -c-n\rightarrow\ t)$

Both directions are proved separately by induction on the operational semantics.

Taking auxiliary variables seriously means that assertions must now depend on them as well as on the state. Initially we do not fix the type of auxiliary variables but parameterize the type of assertions with a type variable $'a$:

types $'a\ assn = 'a \Rightarrow state \Rightarrow bool$

Type constructors are written postfix.

The second major change is the need to reason about Hoare triples in a context: proofs about recursive procedures are conducted by induction where we assume that all *CALL*s satisfy the given pre/postconditions and have to show that the body does as well. The assumption is stored in a context, which is a set of Hoare triples:

types $'a\ cntxt = ('a\ assn \times com \times 'a\ assn)set$

In the presence of only a single procedure the context will always be empty or a singleton set. With multiple procedures, larger sets can arise.

Now that we have contexts, validity becomes more complicated. Ordinary validity (w.r.t. partial correctness) is still what it used to be, except that we have to take auxiliary variables into account as well:

$$\models \{P\}c\{Q\} \ \equiv\ \forall s\ t.\ s\ -c\rightarrow\ t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)$$

Auxiliary variables are always denoted by z.

Validity of a context and validity of a Hoare triple in a context are defined as follows:

$$\begin{array}{rcl} |\models C & \equiv & \forall (P,c,Q) \in C.\ \models \{P\}c\{Q\} \\ C \models \{P\}c\{Q\} & \equiv & |\models C \longrightarrow \models \{P\}c\{Q\} \end{array}$$

Note that $\{\} \models \{P\}\ c\ \{Q\}$ is equivalent to $\models \{P\}\ c\ \{Q\}$.

Unfortunately, this is not the end of it. As we have two semantics, $-c\rightarrow$ and $-c-n\rightarrow$, we also need a second notion of validity parameterized with the recursion depth n:

$$\begin{array}{rcl} \models n\ \{P\}c\{Q\} & \equiv & \forall s\ t.\ s\ -c-n\rightarrow\ t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t) \\ |\models\text{-}n\ C & \equiv & \forall (P,c,Q) \in C.\ \models n\ \{P\}c\{Q\} \\ C \models n\ \{P\}c\{Q\} & \equiv & |\models\text{-}n\ C \longrightarrow \models n\ \{P\}c\{Q\} \end{array}$$

Finally we come to the proof system for deriving triples in a context:

$$C \vdash \{\lambda z\ s.\ P\ z\ (f\ s)\}\ Do\ f\ \{P\}$$

$$[\![\ C \vdash \{P\}c1\{Q\};\ C \vdash \{Q\}c2\{R\}\ ]\!] \Longrightarrow C \vdash \{P\}\ c1;c2\ \{R\}$$

$$[\![\ C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}c1\{Q\};\ \ C \vdash \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}c2\{Q\}\ \ ]\!]$$

$$\Longrightarrow C \vdash \{P\}\ \mathit{IF}\ b\ \mathit{THEN}\ c1\ \mathit{ELSE}\ c2\ \{Q\}$$

$$C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}\ c\ \{P\}$$
$$\Longrightarrow C \vdash \{P\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\}$$

$$[\![\ C \vdash \{P'\}c\{Q'\};\ \forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)\ ]\!]$$
$$\Longrightarrow C \vdash \{P\}c\{Q\}$$

$$\{(P,\mathit{CALL},Q)\} \vdash \{P\}\mathit{body}\{Q\} \Longrightarrow \{\} \vdash \{P\}\ \mathit{CALL}\ \{Q\}$$

$$\{(P,\mathit{CALL},Q)\} \vdash \{P\}\ \mathit{CALL}\ \{Q\}$$

The first four rules are familiar, except for their adaptation to auxiliary variables. The *CALL* rule embodies induction and has already been motivated above. Note that it is only applicable if the context is empty. This shows that we never need nested induction. For the same reason the assumption rule (the last rule) is stated with just a singleton context.

The only real surprise is the rule of consequence, which appears in print in this form for the first time as far as I am aware. Morris [14] and later Olderog [25] show that the consequence rule with side condition

$$\forall s.\ P\ z\ s \longrightarrow (\forall t.\ (\forall z'.\ P'\ z'\ s \longrightarrow Q'\ z'\ t) \longrightarrow Q\ z\ t)$$

(which is already close to our side condition) is "adaption complete" for partial correctness. Olderog also shows completeness of a proof system based on this rule. Hofmann [6] builds on [27] and shows soundness and completeness of a Hoare logic where the consequence rule has the following side condition:

$$\forall s\ t\ z.\ P\ z\ s \longrightarrow Q\ z\ t \vee (\exists z'.\ P'\ z'\ s \wedge (Q'\ z'\ t \longrightarrow Q\ z\ t))$$

The side conditions by Morris and Hofmann are logically equivalent to ours. But the symmetry of our new version appeals not just for aesthetic reasons but because one can actually remember it! Furthermore, its soundness proof is very direct: In order to show $C \models \{P\}\ c\ \{Q\}$ we assume the validity of C (which implies $\models \{P'\}\ c\ \{Q'\}$ because of $C \models \{P'\}\ c\ \{Q'\}$) and prove $\models \{P\}\ c\ \{Q\}$: assuming $s\ -c\rightarrow\ t$, $\models \{P'\}\ c\ \{Q'\}$ implies $\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t$, which, by the side condition of the consequence rule, implies $\forall z.\ P\ z\ s \longrightarrow Q\ z\ t$, which is exactly what we need for $\models \{P\}\ c\ \{Q\}$.

The proof of the soundness theorem

theorem $C \vdash \{P\}c\{Q\} \implies C \models \{P\}c\{Q\}$

requires a generalization: $\forall n.\ C \models n\ \{P\}\ c\ \{Q\}$ is proved instead, from which the actual theorem follows directly via lemma *exec-iff-execn* in Sect. 5.1.

The generalization is proved by induction on c. The reason for the generalization is that soundness of the *CALL* rule is proved by induction on the maximal call depth, i.e. n.

The completeness proof is quite different from the ones we have seen so far. It employs the notion of a *most general triple* (or *most general formula*) due to Gorelick [5]:

$$MGT :: com \Rightarrow state\ assn \times com \times state\ assn$$
$$MGT\ c \equiv (\lambda z\ s.\ z = s,\ c,\ \lambda z\ t.\ z\ -c\rightarrow\ t)$$

Note that the type of z has been identified with *state*. This means that for every state variable there is an auxiliary variable, which is simply there to record the value of the program variables before execution of a command. This is exactly what, for example, VDM offers by allowing you to refer to the pre-value of a variable in a postcondition [11]. The intuition behind $MGT\ c$ is that it completely describes the operational behaviour of c. It is easy to see that, in the presence of the new consequence rule, $\{\} \vdash MGT\ c$ implies completeness:

lemma *MGT-implies-complete*:
$\{\} \vdash MGT\ c \implies \{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q{::}state\ assn\}$

Note that the type constraint $Q{::}state\ assn$ is not inferred automatically: although both pre and postcondition of $MGT\ c$ are of type *state assn*, this does not force Q to have the same type.

In order to discharge $\{\} \vdash MGT\ c$ one proves

lemma *MGT-lemma*: $C \vdash MGT\ CALL \implies C \vdash MGT\ c$

The proof is by induction on c. In the *While*-case it is easy to show that $\lambda z\ t.\ (z,\ t) \in \{(s,\ t).\ b\ s \wedge s\ -c\rightarrow\ t\}^*$ is invariant. The precondition $\lambda z\ s.\ z{=}s$ establishes the invariant and a reflexive transitive closure induction shows that the invariant conjoined with $\neg\ b\ t$ implies the postcondition $\lambda z\ t.\ z\ -WHILE\ b\ DO\ c\rightarrow\ t$. The remaining cases are trivial.

Using the *MGT-lemma* (together with the *CALL* and the assumption rule) one can easily derive

lemma $\{\} \vdash MGT\ CALL$

Using the *MGT-lemma* once more we obtain $\{\} \vdash MGT\ c$ and thus by *MGT-implies-complete* completeness.

theorem $\{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q{::}state\ assn\}$

5.3. MODULAR EXTENSIONS

Procedures can be extended with nondeterministic choice and local variables just as we extended the pure while-language in Sect. 2.2: the operational semantics is the same, and the Hoare rules just need to be extended with a context C. The proofs are identical as well. But in the case of local variables the resulting language may not be what one expects: it has the semantics of *dynamic scoping*: if the procedure body refers to some variable, say x, then the execution of that body during the execution of $VAR\ x = e;\ CALL$ will refer to the local x with initial value e. We have also studied a language with static scoping, but do not discuss the details in this paper.

5.4. HOARE LOGIC FOR TOTAL CORRECTNESS

This is the most complicated system in this paper. We only show the key elements and none of the proofs.

For termination we have just one new obvious rule:

$$body \downarrow s \Longrightarrow CALL \downarrow s$$

Validity is defined as expected:

$$\begin{aligned} \models_t \{P\}c\{Q\} &\equiv \models \{P\}c\{Q\} \wedge (\forall z\ s.\ P\ z\ s \longrightarrow c{\downarrow}s) \\ C \models_t \{P\}c\{Q\} &\equiv (\forall (P',c',Q') \in C.\ \models_t \{P'\}c'\{Q'\}) \longrightarrow \models_t \{P\}c\{Q\} \end{aligned}$$

Instead of the full set of proof rules we merely show those that differ from the system for partial correctness:

$$\begin{aligned} &[\![wf\ r;\ \forall s'.\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge b\ s \wedge s' = s\}\ c\ \{\lambda z\ s.\ P\ z\ s \wedge (s,s') \in r\}]\!] \\ &\Longrightarrow C \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg b\ s\} \end{aligned}$$

$$\begin{aligned} &[\![wf\ r;\ \forall s'.\ \{(\lambda z\ s.\ P\ z\ s \wedge (s,s') \in r,\ CALL,\ Q)\} \\ &\qquad \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge s = s'\}\ body\ \{Q\}]\!] \\ &\Longrightarrow \{\} \vdash_t \{P\}\ CALL\ \{Q\} \end{aligned}$$

$$\begin{aligned} &[\![\ C \vdash_t \{P'\}c\{Q'\}; \\ &\quad (\forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)) \wedge \\ &\quad (\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s))\ ]\!] \\ &\Longrightarrow C \vdash_t \{P\}c\{Q\} \end{aligned}$$

As in the case for the pure while-language, our rules for total correctness are very similar to those by Kleymann [11]. The side condition in our rule of consequence looks quite different from the one by Kleymann, but the two are in fact equivalent:

lemma $((\forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)) \wedge$
$(\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s)))$
$= (\forall z\ s.\ P\ z\ s \longrightarrow (\forall t. \exists z'.\ P'\ z'\ s \wedge (Q'\ z'\ t \longrightarrow Q\ z\ t)))$

Kleymann's version is easier to use, whereas our new version clearly shows that it is a conjunction of the side condition for partial correctness with precondition strengthening.

The key difference to the work by Kleymann (and America and de Boer) is that soundness and completeness

theorem $C \vdash_t \{P\}c\{Q\} \implies C \models_t \{P\}c\{Q\}$
theorem $\{\} \models_t \{P\}c\{Q\} \implies \{\} \vdash_t \{P\}c\{Q{::}state\ assn\}$

are shown for arbitrary, i.e. unbounded nondeterminism. This is a significant extension and appears to have been an open problem. The details are found in a separate paper [18].

6. More procedures

We now generalize from a single procedure to a whole set of procedures following the ideas of von Oheimb [21]. The basic setup of Sect. 5.1 is modified only in a few places:

- We introduce a new basic type *pname* of procedure names.
- Constant *body* is now of type $pname \Rightarrow com$.
- The *CALL* command now has an argument of type *pname*, the name of the procedure that is to be called.
- The call rule of the operational semantics now says

$$s\ -body\ p\rightarrow\ t \Longrightarrow s\ -CALL\ p\rightarrow\ t$$

Note that this setup assumes that we have a procedure body for each procedure name. If you feel uncomfortable with the idea of an infinity of procedures, you may assume that *pname* is the finite subset of all procedure names that occur in some fixed program.

6.1. HOARE LOGIC

Types *assn* and and *cntxt* are defined as in Sect. 5.2, as are $\models \{P\}\ c\ \{Q\}$, $|\models C$, $\models n\ \{P\}\ c\ \{Q\}$ and $|\models$-$n\ C$. However, we now need an additional notion of validity $C\ |\models D$ where D is a set as well. The reason is that we can now have mutually recursive procedures whose correctness needs to be established by simultaneous induction. Instead of sets of Hoare triples

we may think of conjunctions. We define both $C \mid\models D$ and its relativized version:

$$C \mid\models D \quad \equiv \quad \mid\models C \longrightarrow \mid\models D$$
$$C \mid\models\text{-}n\ D \ \equiv \ \mid\models\text{-}n\ C \longrightarrow \mid\models\text{-}n\ D$$

Our Hoare logic now defines judgements of the form $C \Vdash D$ where both C and D are (potentially infinite) sets of Hoare triples; $C \vdash \{P\}\ c\ \{Q\}$ is simply an abbreviation for $C \Vdash \{(P,c,Q)\}$. With this abbreviation the rules for *Do*, *Semi*, *If*, *While* and consequence are exactly the same as in Sect. 5.2. The remaining rules are

$$[\![\ \forall (P,c,Q) \in C.\ \exists p.\ c = CALL\ p;$$
$$\quad C \Vdash \{(P,b,Q).\ \exists p.\ (P,CALL\ p,Q) \in C \wedge b = body\ p\}\ ]\!]$$
$$\Longrightarrow \{\} \Vdash C$$

$$(P,CALL\ p,Q) \in C \Longrightarrow C \vdash \{P\}\ CALL\ p\ \{Q\}$$

$$\forall (P,c,Q) \in D.\ C \vdash \{P\}c\{Q\} \implies C \Vdash D$$
$$[\![\ C \Vdash D;\ (P,c,Q) \in D\ ]\!] \Longrightarrow C \vdash \{P\}c\{Q\}$$

The *CALL* and the assumption rule are straightforward generalizations of their counterparts in Sect. 5.2. The final two rules are structural rules and could be called conjunction introduction and elimination, because they put together and take apart sets of triples.

theorem $C \Vdash D \implies C \mid\models D$

As before, we prove a generalization of $C \mid\models D$, namely $\forall n.\ C \mid\models\text{-}n\ D$, by induction on $C \Vdash D$, with an induction on n in the *CALL* case.

The completeness proof resembles the one in Sect. 5.2 closely: the most general triple *MGT* is defined exactly as before, and the lemmas leading up to completness are simple generalizations:

lemma *MGT-implies-complete*:
$\{\} \Vdash \{MGT\ c\} \Longrightarrow \models \{P\}c\{Q\} \Longrightarrow \{\} \vdash \{P\}c\{Q::state\ assn\}$
lemma *MGT-lemma*: $\forall p.\ C \Vdash \{MGT(CALL\ p)\} \implies C \Vdash \{MGT\ c\}$
lemma $\{\} \Vdash \{mgt.\ \exists p.\ mgt = MGT(CALL\ p)\}$
theorem $\models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q::state\ assn\}$

7. Functions

As our final variation on the theme of procedures we study a language with functions, i.e. procedures that take arguments and return results. Our functions have exactly one parameter which is passed by value.

7.1. SYNTAX AND OPERATIONAL SEMANTICS

The basic types *var*, *val*, *state* and *bexp* are defined as in Sect. 2. In addition we have

types *expr* $=$ *state* $\Rightarrow$ *val*

Commands are also defined as in Sect. 5.1, but extended with function calls and local variables:

| *Fun var expr*	(- := *FUN* - *60*)
| *Var var expr com*	(*VAR* - = -; -)

Command *Var* is familar. Command $x := FUN\ e$ is meant to pass the value of e as the (single) parameter to the (single) function in the program and to assign the result of the function to x. This syntax rules out nested expressions and thus all the complications treated in Sect. 4. The body of the function is again found in constant *body*. Furthermore there are two distinguished variables *arg*, the name of the formal parameter, and *res*, a variable for communicating the result back to the caller. Function calls are reduced to procedure calls as follows:

$$funcall\ x\ e \equiv (VAR\ arg = e;\ CALL);\ Do(\lambda s.\ s(x := s\ res))$$

The argument becomes a local variable and is thus not modified by further function calls in the body. Variable *res* is not protected like this and it is the programmer's responsibility to introduce a local variable if needed.

Note that *CALL* is still present to allow this two stage execution of function calls but it is not meant to be used on its own.

The semantics of *Fun*, the only new command, is obvious:

$$s -funcall\ x\ e\rightarrow t \implies s -x := FUN\ e\rightarrow t$$

The rule for local variables is the same as in Sect. 5.2, i.e. we have dynamic scoping, which is just what we need for the above implementation of parameter passing.

In essence, this is all there is to say about functions, because we have reduced function calls to the composition of previously studied language constructs.

7.2. HOARE LOGIC

Assertions and validity are defined as for procedures. The proof system is the one from Sect. 5.2 together with the rule for *Var* discussed in Sect. 5.2, extended with the obvious rule for function calls:

$C \vdash \{P\}funcall\ x\ e\{Q\} \Longrightarrow C \vdash \{P\}\ x := FUN\ e\ \{Q\}$

Soundness and completeness are proved entirely as for procedures.

theorem $C \vdash \{P\}c\{Q\} \implies C \models \{P\}c\{Q\}$
theorem $\{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q::state\ assn\}$

8. Conclusion

The preceding sections show that a variety of language constructs can be given a simple Hoare logic in the unified framework of HOL. In particular we have been able to settle the case of total correctness of recursive procedures combined with unbounded nondeterminism [18].

Although one would think that Hoare logics formalized in theorem provers should always come with a completeness proof, this is not so. The argument used by [8] to explain the absence of a completeness proof is the ability to fall back on a denotational semantics if the Hoare logic fails. The authors of [13] even claim that completeness is not a meaningful question in their setting. This paper has shown that completeness can be established by standard means even for complicated language constructs. Thus it should not be ignored.

Acknowledgments I am indebted to Thomas Kleymann and David von Oheimb for providing the logical foundations and to Markus Wenzel for the Isar extension of Isabelle: without it the production of the paper from the Isabelle theories, something I would no longer want to miss, would have been impossible. Gerwin Klein, Norbert Schirmer and Markus Wenzel commented on a draft version.

References

1. Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84:129–162, 1990.
2. Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part I. *ACM Trans. Programming Languages and Systems*, 3(4):431–483, 1981.
3. Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.
4. Krzysztof Apt and Lambert Meertens. Completeness with finite systems of intermediate assertions for recursive program schemes. *SIAM Jornal on Computing*, 9(4):665–671, 1980.
5. Gerald Arthur Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dept. of Computer Science, Univ. of Toronto, 1975.
6. Martin Hofmann. Semantik und Verifikation. Lecture notes, Universität Marburg. In German, 1997.

7. Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lect. Notes in Comp. Sci.*, pages 201–215. Springer-Verlag, 1996.
8. Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lect. Notes in Comp. Sci.*, pages 284–299. Springer-Verlag, 2001.
9. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990.
10. Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, Department of Computer Science, University of Edinburgh, 1998. Report ECS-LFCS-98-392.
11. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11:541–566, 1999.
12. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.
13. Linas Laibinis and Joakim von Wright. Functional procedures in higher-order logic. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1896 of *Lect. Notes in Comp. Sci.*, pages 372–387. Springer-Verlag, 2000.
14. J.H. Morris. Comments on "procedures and parameters". Undated and unpublished.
15. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley, 1992.
16. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
17. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
18. Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. Draft, 2001.
19. Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170, 1998.
20. Tobias Nipkow and Lawrence Paulson. *Isabelle/HOL. The Tutorial*, 2001. http://www.in.tum.de/~nipkow/pubs/tutorial.html.
21. David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 168–180. Springer-Verlag, 1999.
22. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. http://www.in.tum.de/~oheimb/diss/.
23. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.
24. David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. Submitted for publication, 2001.
25. Ernst-Rdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.

26. Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
27. Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 697–711. Springer-Verlag, 1997.