Proving Properties on Programs
—From the Coq Tutorial at ITP 2015—

Reynald Affeldt

August 29, 2015

Hoare logic is a proof system to verify imperative programs. It consists of a language of Hoare triples $\{P\}c\{Q\}$ where c is a program, P is a pre-condition, and Q is a post-condition, that the execution of a program is supposed to respect.

Here, we provide a Hoare logic for a simple imperative language (while-loops and parameterless procedure calls) and apply it to the verification of programs computing the factorial function. The main reference is [1].

This lecture introduces examples of inductive data types and predicates (following the previous lectures). It provides a simple example of Coq modules. We rely only on Coq's standard library. In particular, we use finite maps and sets (in the sense of `Ensemble`).

Overview We first define a functor with a generic Hoare logic: syntax of the language in Sect. 1.2, operational semantics in Sect. 1.3, and Hoare logic in Sect. 1.4. We state the soundness of Hoare logic in Sect. 1.5. We instantiate this module with the basic instruction of variable assignment in Sect. 2.1, and apply the result to two proofs of the factorial program: one using a while-loop (Sect. 2.2) and one using recursive procedure calls (Sect. 2.3).

Contents

About the Exercise The student is expected to complete the holes in the various proofs of the companion Coq file. It is maybe more interesting to start with the example in module `FactorialWhile`, then the one in module `FactorialRec`. All the intermediate assertions are provided. Afterwards, move to the proofs in functor `Cmd`. Most proofs are simple inductions w.r.t. the operational semantics; the main step and/or hints are provided so that one can just exercise with Coq tactics. The soundness proof is non-trivial and can be done last. The main induction principle is provided.

1 A Generic Hoare Logic

1.1 The Setting

We assume the following abstract type `state`, wrapped in the following interface:

```
Module Type STATE.

Parameter state : Set.
Definition assert := state → Prop.
Definition entails (P Q : assert) : Prop := ∀ s, P s → Q s.
Parameter exp : Set.
Parameter bexp : Set.
Parameter eval : exp → state → nat.
Parameter beval : bexp → state → Prop.

End STATE.
```

Functions of type `assert` are used to specify states in pre/post-conditions of Hoare logic. All the standard logical connectives can be lifted from `Prop` to `assert`. In particular, `entails` defines the lifting of the Coq implication → to `assert` and it is denoted by `===>`.

`exp` is intended to be the type of arithmetic expression and `bexp` is intended to be the type of boolean expressions. `eval` (resp. `beval`) is an evaluation function for expressions in a given state and is denoted by `[ e ]_s` (resp. `[ b ]b_s`). We do not need a concrete definition at this point.

We also assume the following abstract type `cmd0` for basic commands (such as assignment, memory lookup, etc.):

```
Module Type CMD0 (S : STATE).

Parameter cmd0 : Set.
Parameter exec0 : S.state → cmd0 → option S.state → Prop.
Parameter hoare0 : S.assert → cmd0 → S.assert → Prop.

End CMD0.
```

The ternary relation `exec0` is the operational semantics. A `None` state represents an execution error. The ternary relation `hoare0` is intended to be the corresponding Hoare logic. We do not need concrete definitons at this point.

Our functor for Hoare logic start with the declaration

```
Module Cmd (S : STATE) (Cmd0 : CMD0 S).
...
```

whose body is the topic of the next sections.

1.2 Syntax

The language we are dealing with is Winskel's While language extended with parameterless procedures. We first provide a type for procedure names. Since they will be used as an environment for the semantics, we use the finite maps from the Coq standard library:

```
Module Procs := FMapList.Make (String_as_OT).
Definition proc := Procs.key.
```

In this case, `proc` is just `string`.

The BNF for the language is unsurprising:

$$
\begin{array}{rcl}
c & ::= & \texttt{basic}\ c_0 \\
 & | & c_1\ \texttt{.;}\ c_2 \\
 & | & \texttt{If}\ b\ \texttt{Then}\ c_1\ \texttt{Else}\ c_2 \\
 & | & \texttt{while}\ b\ c' \\
 & | & \texttt{call}\ p
\end{array}
$$

It is encoded simply as an inductive type, each production rule taking the form of a constructor:

```
Inductive cmd : Type :=
| basic : Cmd0.cmd0 → cmd
| seq : cmd → cmd → cmd
| ifte : bexp → cmd → cmd → cmd
| while : bexp → cmd → cmd
| call : proc → cmd.
```

In the following, sequences are noted by .; and conditional branching is denoted by `If ... Then ... Else`.

1.3 Operational Semantics

The operational semantics is defined by a ternary predicate `exec` of type

```
state → cmd → option state → Prop
```

with a parameter, say l, for the environment of procedures. It is denoted by l `|~` s `>-` c `--->` s': s is state before execution of c, and s' is the state after execution.

The environment of procedures is a map from procedure names to their bodies, i.e., from type `proc` to type `cmd`:

```
Definition procs := Procs.t cmd.
```

The operational semantics is syntax-oriented: it is defined by an inductive predicate and for each syntactic construct of the language there is one or two constructors (depending on branching behavior or error states). Let us first lookup at the execution rules in their traditional, pencil-and-paper presentation:

$$\frac{\texttt{exec0}\ s\ c_0\ s_1}{l\ \texttt{|}{\sim}s\ \texttt{>-}\ \texttt{basic}\ c_0\ \texttt{--->}\ s_1}\ \texttt{exec\_basic}$$

$$\frac{l\ \texttt{|}{\sim}s\ \texttt{>-}\ c\ \texttt{--->}\ \texttt{Some}\ s_1 \qquad l\ \texttt{|}{\sim}s_1\ \texttt{>-}\ d\ \texttt{--->}\ s_2}{l\ \texttt{|}{\sim}s\ \texttt{>-}\ c\ \texttt{.;}\ d\ \texttt{--->}\ s_2}\ \texttt{exec\_seq}$$

$$\frac{\texttt{[}b\texttt{]b\_}s \qquad l\ \texttt{|}{\sim}s\ \texttt{>-}\ c\ \texttt{--->}\ s_1}{l\ \texttt{|}{\sim}s\ \texttt{>-}\ \texttt{If}\ b\ \texttt{Then}\ c\ \texttt{Else}\ d\ \texttt{--->}\ s_1}\ \texttt{exec\_ifte\_true}$$

$$\frac{\begin{array}{c}\texttt{[}b\texttt{]b\_}s \qquad l\ \texttt{|}{\sim}s\ \texttt{>-}\ c\ \texttt{--->}\ \texttt{Some}\ s_1 \\ l\ \texttt{|}{\sim}s_1\ \texttt{>-}\ \texttt{while}\ b\ c\ \texttt{--->}\ s_2\end{array}}{l\ \texttt{|}{\sim}s\ \texttt{>-}\ \texttt{while}\ b\ c\ \texttt{--->}\ s_2}\ \texttt{while\_true}$$

$$\frac{l \vdash p \mapsto c \qquad l\ \texttt{|}{\sim}s\ \texttt{>-}\ c\ \texttt{--->}\ s_1}{l\ \texttt{|}{\sim}s\ \texttt{>-}\ \texttt{call}\ p\ \texttt{--->}\ s_1}\ \texttt{exec\_call}$$

The Coq formalization of the above operational semantics is direct:

```
Reserved Notation "l |~ s ≻— c ⟶ t".
Inductive exec (l : procs) : state → cmd → option state → Prop :=
| exec_basic : ∀ s c0 s1, Cmd0.exec0 s c0 s1 →
  l |~ s ≻— basic c0 ⟶ s1
| exec_seq : ∀ s s1 s2 c d,
  l |~ s ≻— c ⟶ Some s1 → l |~ s1 ≻— d ⟶ s2 →
  l |~ s ≻— c .; d ⟶ s2
| exec_ifte_true : ∀ s s1 b c d, [ b ]b_ s →
  l |~ s ≻— c ⟶ s1 → l |~ s ≻— If b Then c Else d ⟶ s1
| exec_ifte_false : ∀ s s1 b c d, ~ [ b ]b_ s →
  l |~ s ≻— d ⟶ s1 → l |~ s ≻— If b Then c Else d ⟶ s1
| exec_while_true : ∀ s s1 s2 b c,
  [ b ]b_ s → l |~ s ≻— c ⟶ Some s1 →
  l |~ s1 ≻— while b c ⟶ s2 → l |~ s ≻— while b c ⟶ s2
| exec_while_false : ∀ s b c,
  ~ [ b ]b_ s → l |~ s ≻— while b c ⟶ Some s
| exec_call : ∀ s s1 p c, Procs.MapsTo p c l →
  l |~ s ≻— c ⟶ s1 → l |~ s ≻— call p ⟶ s1
| exec_call_err : ∀ s p,
  Procs.find p l = None →
  l |~ s ≻— call p ⟶ None
where "l |~ s ≻— c ⟶ t" := (exec l s c t).
```

1.4 Hoare Logic

Since we are dealing with procedures, the Hoare logic takes, like the operational semantics, the procedure environment as a parameter. Moreover, it is indexed by the set of "procedure specifications" that we can assume to hold; this makes it possible in particular to handle recursive calls.

The procedure specifications are essentially Hoare triples:

```
Record spec := Spec {
  pre : assert ;
  callee : proc ;
  post : assert}.
```

Sets of procedure specifications are represented by the type `Ensembles.Ensemble` from the Coq standard library.

In summary, the Hoare logic is defined as a quaternary predicate of type `Ensemble spec` $\to$`assert` $\to$`cmd` $\to$`assert` $\to$`Prop` with parameter of type `procs`.

The rules for sequence, conditional branching, and while-loops are textbook.

$$\frac{\texttt{hoare0}\ P\ c\ Q}{l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P\,\texttt{]\}}\,\texttt{basic}\ c\,\texttt{\{[}\,Q\,\texttt{]\}}}\ \text{hoare\_basic}$$

$$\frac{\texttt{l \textbackslash\^{} E |\~{}\{[P]\} c \{[Q]\}} \qquad \texttt{l \textbackslash\^{} E |\~{}\{[Q]\} d \{[R]\}}}{\texttt{l \textbackslash\^{} E |\~{}\{[P]\} c .; d \{[R]\}}}\ \text{hoare\_seq}$$

$$\frac{l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[fun}\ s\ \texttt{=>}\ P\,s \wedge \texttt{[}b\texttt{]b\_}s\texttt{]\}}\,c\,\texttt{\{[}\,P\,\texttt{]\}}}{l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P\,\texttt{]\}}\,\texttt{while}\ b\ c\,\texttt{\{[fun}\ s\ \texttt{=>}\ P\,s \wedge \texttt{\~{}[}b\texttt{]b\_}s\texttt{]\}}}\ \text{hoare\_while}$$

There are two rule for procedure calls. hoare_call2 is for when the Hoare triple happens to be a specification from the environment. hoare_call makes it possible to handle recursive calls: it extends the environment of procedure specifications E with new specifications E' in which one may assume the conclusion Hoare triple.

$$\frac{\texttt{Spec}\,P\,p\,Q \in E}{l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P\,\texttt{]\}}\,\texttt{call}\ p\,\texttt{\{[}\,Q\,\texttt{]\}}}\ \text{hoare\_call2}$$

$$\frac{\forall E', \texttt{Spec}\,P\,p\,Q \in E' \qquad \begin{array}{l}\forall t, t \in E' \to \exists c, l \vdash \texttt{callee}\ t \mapsto c\ \wedge \\ l \texttt{\textbackslash\^{}}\ E \cup E'\ \texttt{|\~{}\{[pre}\ t\texttt{]\}}\,c\,\texttt{\{[post}\ t\texttt{]\}}\end{array}}{l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P\,\texttt{]\}}\,\texttt{call}\ p\,\texttt{\{[}\,Q\,\texttt{]\}}}\ \text{hoare\_call}$$

The textbook consequence rule would be the following one:

$$\frac{P \texttt{ ===> } P' \qquad l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P'\,\texttt{]\}}\,c\,\texttt{\{[}\,Q'\,\texttt{]\}} \qquad Q' \texttt{ ===> } Q}{l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P\,\texttt{]\}}\,c\,\texttt{\{[}\,Q\,\texttt{]\}}}$$

We actually need to generalize it so as to handle recursion. The following hoare_conseq rule makes it possible to transport information about auxiliary variables from the pre-condition to the post-condition (see exercise):

$$\frac{\forall s, P\,s \to \exists P'Q', l \texttt{\textbackslash\^{}}\ E\ \texttt{|\~{}\{[}\,P'\,\texttt{]\}}\,c\,\texttt{\{[}\,Q'\,\texttt{]\}} \wedge P'\,s \wedge Q' \texttt{ ===> } Q}{\texttt{l \textbackslash\^{} E |\~{}\{[P]\} c \{[Q]\}}}\ \text{hoare\_conseq}$$

The encoding of the corresponding inductive predicate follows the pencil-and-paper description:

```
Reserved Notation "l \^ E |~{[ P ]} c {[ Q ]}".
Inductive hoare (l : procs) : Ensemble spec → assert → cmd → assert →
Prop :=
| hoare_basic : ∀ E P Q c, Cmd0.hoare0 P c Q →
  l \^ E |~{[ P ]} basic c {[ Q ]}
| hoare_seq : ∀ E P Q R c d,
  l \^ E |~{[ P ]} c {[ Q ]} → l \^ E |~{[ Q ]} d {[ R ]} →
  l \^ E |~{[ P ]} c .; d {[ R ]}
| hoare_conseq : ∀ E c (P Q : assert),
  (∀ s, P s → ∃ P' Q',
    l \^ E |~{[ P' ]} c {[ Q' ]} ∧ P' s ∧ (Q' ⟹ Q)) →
  l \^ E |~{[ P ]} c {[ Q ]}
| hoare_while : ∀ E P b c,
  l \^ E |~{[ fun s ⇒ P s ∧ [ b ]b_ s ]} c {[ P ]} →
  l \^ E |~{[ P ]} while b c {[ fun s ⇒ P s ∧ ~ [ b ]b_ s ]}
| hoare_ifte : ∀ E P Q b c d,
  l \^ E |~{[ fun s ⇒ P s ∧ [ b ]b_ s ]} c {[ Q ]} →
  l \^ E |~{[ fun s ⇒ P s ∧ ~ [ b ]b_ s ]} d {[ Q ]} →
  l \^ E |~{[ P ]} If b Then c Else d {[ Q ]}
| hoare_call : ∀ E P Q p E',
  In E' (Spec P p Q) →
  (∀ t, In E' t → ∃ c, Procs.MapsTo (callee t) c l ∧
     l \^ Union E E' |~{[ pre t ]} c {[ post t ]}) →
  l \^ E |~{[ P ]} call p {[ Q ]}
| hoare_call2 : ∀ E P Q p, In E (Spec P p Q) →
  l \^ E |~{[ P ]} call p {[ Q ]}
where "l \^ E |~{[ P ]} c {[ Q ]}" := (hoare l E P c Q).
```

1.5 Soundness of Hoare Logic

We have to check that the proof system of Sect. 1.4 agrees with the operational semantics of Sect. 1.3. For that purpose, we define the meaning of Hoare triples using the operational semantics. We denote by $l \backslash\hat{}\ E$ `|={{` P `}}` c `{{` Q `}}` this semantics of Hoare triples. It is defined as follows. First, the semantics without the environment of specifications:

```
Definition hoare_sem l P c (Q : assert) : Prop :=
  ∀ s, P s → ~ l |~ s >- c ⟶ None ∧
    (∀ s1, l |~ s >- c ⟶ Some s1 → Q s1).
Notation "l |={{ P }} c {{ Q }}" := (hoare_sem l P c Q).
```

Second, we take care of all the specification in the environment:

```
Definition hoare_sem_ctxt l E P c Q :=
  (∀ t, In E t → l |={{ pre t }} call (callee t) {{ post t }}) →
  l |={{ P }} c {{ Q }}.
Notation "l \^ E |={{ P }} c {{ Q }}" := (hoare_sem_ctxt l E P c Q)
```

Proving the soundness of Hoare logic amounts to prove the following lemma, which of course takes as hypothesis the soundness of not-yet-instantiated basic commands):

```
Lemma hoare_sound E P Q l c :
  (∀ P Q l c, Cmd0.hoare0 P c Q → l |={{ P }} basic c {{ Q }}) →
  l \^ E |~{[ P ]} c {[ Q ]} → l \^ E |={{ P }} c {{ Q }}.
```

The proof uses intermediate lemmas that require an operational semantics augmented with a natural `n` that counts the depth of procedure calls. See the companion Coq file for details.

2 Verification of Concrete Imperative Programs

We now instantiate the generic Hoare logic of Sect. 1, and use it to verify imperative implementations of the factorial function.

2.1 Instantiation

We instantiate states as maps from variables (represented by strings) to natural numbers:

```
Module Vars := FMapList.Make (String_as_OT).
Definition var := Vars.key.
...
Definition state := Vars.t nat.
```

We provide a concrete arithmetic language `exp`:

```
Inductive exp :=
| exp_var : var → exp
| add : exp → exp → exp
| mul : exp → exp → exp
| sub : exp → exp → exp
| cst : nat → exp.
```

In particular, a variable is denoted by % `v`. Addition (resp. multiplication and subtraction) are denoted by `\+` (resp. `\*` and `\-`).

The concrete boolean language `bexp` is built on top of `exp`:

```
Inductive bexp :=
| equa : exp → exp → bexp
| neg : bexp → bexp.
```

We note `\=` for boolean equality and `\!=` for inequality (i.e., a combination of `neg` and `equa`).

Last, we provide a syntax, semantics, and Hoare triple for the assignment command.

Assignment is denoted by $v \leftarrow e$ where e is an expression and v a variable:

```
Inductive cmd0 :=
| assign : var → exp → cmd0.
```

The operational semantics reads as $\emptyset$ `|~`s `>-` $v \leftarrow e$ `--->` $s\{[e]\_s/v\}$:

```
Inductive exec0 : state → cmd0 → option state → Prop :=
| exec0_assign : ∀ s v e, exec0 s (v ← e) (Some (upd v ([ e ]_s) s)).
```

The update of variables is implemented by updating the state by destructive addition:

```
Definition upd v (n : nat) s := Vars.add v n s.
```

The Hoare triple reads as $\emptyset$ `\^` $\emptyset$ `|~{[`$Q\{[e]\_s/v\}$`]}` $v \leftarrow e$ `{[`Q`]}`.

```
Inductive hoare0 : assert → cmd0 → assert → Prop :=
| hoare0_assign : ∀ Q v e, hoare0 (wp_assign v e Q) (v ← e) Q.
```

The substitution is implemented by the predicate transformer `wp_assign` whose semantics is also defined using state update:

```
Inductive wp_assign v e P : assert :=
| wp_assign_c : ∀ s, P (upd v ([ e ]_s) s) → wp_assign v e P s.
```

We can now instantiate the functor of Sect. 1 with concrete states and basic commands:

```
Module C := Cmd state cmd0.
```

This provides types `C.cmd` and `C.hoare` that can be used to perform verification of While programs with parameterless procedures. We also have the soundness guarantee since the following holds (for the `cmd0` type):

```
Lemma sound0 P Q l c : cmd0.hoare0 P c Q → l |={{ P }} C.basic c {{ Q }}.
```

2.2 Factorial as a While-loop

The following C-like factorial program

```
while (x != 0) {
 ret = ret * x ;
 x = x - 1
}
```

is written as follows using our Coq encoding:

```
Definition facto :=
  (C.while (% "x" \≠ 0)
    ("ret" ← % "ret" \* % "x" .;
     "x" ← % "x" \- 1))%string.
```

For specification, we use the Gallina function `fact` from the `Factorial` module of the standard library. If `facto` is started in a state where `x` has value `X` and `ret` has value 1, then it will end up in a state where `ret` has value `fact X` ($X!$):

```
Lemma facto_fact X :
  (Procs.empty _ \^
  @Empty_set _
  |~{[ fun s ⇒ [% "x" ]_s = X ∧ [% "ret" ]_s = 1 ]}
  facto
  {[ fun s ⇒ [ % "ret" ]_ s = fact X ]})%string.
```

Here, the environment is empty (since there is no procedure call). The main idea of the proof is to use $ret * x = X!$ for the loop invariant.

2.3 Factorial as a Recursive Function

We now verify another version of factorial written with a recursive call. In a C-like, informal syntax:

```
void facto {
  if x == 0 {
    ret = 1
  } else {
    x = x - 1 ;
    facto ()
    x = x + 1;
    ret = ret * x
  }
}
```

Using our syntax:

```
Definition facto : C.cmd :=
  (If (% "x" \= 0) Then
    "ret" ← 1
  Else
   "x" ← % "x" \- 1 .;
   C.call "facto" .;
   "x" ← % "x" \+ 1 .;
   "ret" ← % "ret" \* % "x")%string.
```

Verification of the recursive `facto` amounts to prove the following Hoare triple. It is similar to the one for the while-version except that the procedure environment contains the procedure body associated to the string "facto":

```
Lemma facto_fact X :
  (Procs.add "facto" facto (Procs.empty _) \^
  @Empty_set _
  |~{[ fun s ⇒ [% "x" ]_s = X ]}
  C.call "facto"
  {[ fun s ⇒ [% "x" ]_s = X ∧ [ % "ret" ]_ s = fact X ]})%string.
```

Proof Overview The proof starts by applying hoare_call with the addition of $\bigcup_{x<X}$ `factospec` x to the environment, where `factospec` is:

```
Definition factospec x := (C.Spec
  (fun s ⇒ [ %"x" ]_s = x)
  "facto"
  (fun s ⇒ [ %"x" ]_s = x ∧ [ % "ret" ]_ s = fact x))%string.
```

Then, when running into the recursive call, the Hoare triple should look like this:

$$l \setminus\hat{}\ \bigcup_{x<X} \texttt{factospec}\ x \ \texttt{|\~{}}\quad \{[\, x = n - 1 \wedge 1 \leq n \,]\}\ \texttt{call}\ facto\ \{[\, x = n - 1 \wedge ret = (n-1)! \wedge 1 \leq n \,]\}$$

. We need to adapt the pre/post-conditions so as to apply hoare_call2. We use the hoare_conseq rule for that purpose.

See the companion Coq file for the sketch of formal proofs.

References

[1] Norbert Schirmer. Verification of Sequential Imperative Programs in Isabelle/HOL. Technische Universität München. 2006.