ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dipl.-Inf. Achim D. Brucker
Dr. Burkhart Wolff

Computer-supported Modeling and Reasoning

http://www.infsec.ethz.ch/education/permanent/csmr/

(rev. 16802)

Submission date: –

HOL: Hoare Logic

With this exercise, we turn now to applications of Isabelle/HOL in the field of (theoretical) computer science. We will reuse an existing encoding of an imperative toy-language for verifications of imperative programs. From the theorem proving side, we will introduce into structured proofs with ISAR.

1 More on Isabelle: Some ISAR Features

1.1 Structured Proofs with ISAR: An Introduction

Interactive theorem proving (as we introduced it in the course and as we — the authors — still believe is easier to understand comprehensively) has been dominated by a model of proof that goes back to the LCF system: a proof is a sequence of commands that manipulate an implicit proof state.

This model is reflected in the syntactic structure:

(**lemma** | theorem) [*name* :] <proposition> <**proof**>

where <**proof**> has is a sequence of **apply**(<method>) commands followed by **done** or just **by**(method).

Tactic-style proofs had been criticized for being very distinct from mathematics-like texts, unstructured and hard to maintain. Therefore, ISAR

has been conceived to allow a more declarative proof-style that is claimed to be closer to mathematical texts (the reader may browse through the meanwhile quite rich corpus of structured proofs in the library in order to decide if this goal has really been achieved).

Structured proofs were introduced by a new alternative in the syntactic category <**proof**> which introduces a block structure:

fix
assume
from
this
show
have
note
let

<**proof**> ::= ...
| **proof** [<method> | −] <**proof**> {<statement>} **qed**

Here, a <statement> has the form:

<statement> ::= **fix** {name}
| **assume** [<fact>:] <propositions>
| [**from** {<fact>} | **this**} (**show** | **have**) <propositions>
<**proof**>
| **note** <fact> = <fact> | **this**
| **let** <meta−var> = "*term*"

and can thus again contain sub-proofs.

In the following, we discuss <statement> in more detail. The <**fix**>-statement serves as abstract means to introduce meta-quantified variables in a local proof goal, the <**assume**>-statement is used (similarly to the <**assumes**>-statement on the top-level) to introduce local assumptions and the <**have**>-statement to introduce the conclusion of a *local subgoal* of a proof. Thus, within a proof, local subgoals can be stated and proven. With the <**note**>-statement, the previous proposition (referenced by <**this**>) can be bound to a name, and in a <**let**> statement, a meta-variable may be bound to a particular *term*; since this meta-variable may be used in subsequent propositions, this may be used to reduce the size of local propositions and substitutions drastically. In connection with a pattern-match construct possible in any:

<proposition> ::= "*term*" [({is "<string>"})]

(where in the string, meta-variables may be used that can also be used in propositions and substitutions later), a means for systematic abbreviations in proof texts is provided.

Note that with the **proof**-directive, the current proof state is implicitly bound to a particular meta-variable ?thesis. Consequently, in order to conclude a subproof successfully, a proof will typically have the form:

```
proof −
  assume "the−assm"
  have   "concl" by (...)
```

```
note   A = this
assume "the−other−asm"
have   "the−other−concl" by(...)
note   B = this
show   ?thesis
       <main proof>
```

Note that the − symbol stands for "do nothing"; if omitted, the default method is application of certain introduction rules controlled by the context.

Obviously, ISAR has been reduced to a kind of core-language here; a large number of abbreviations and syntactic variations exist. For example, there is an implicit fact management (pretty much inspired by PEARL) that makes most **note**-statements superfluous. We will describe some of these variations in subsequent exercise sheets.

2 Exercises

This exercise is based on IMP, in particular VC, which is not a IsabelleHOL built-in. You will need to extend Isabelle's search path such that Isabelle will be able to load the needed theory files at run-time. Therefore, start your theory file like:

```
ML {*
  add_path "$ISABELLE_HOME/src/HOL/IMP";
*}

theory HOL_Hoare = VC:
```

2.1 Exercise 43

Verify the program for computing the integer square root (from the lecture) in IMP from the lecture:

```
((  tm  :== (λs. 1));
 ((sum  :== (λs. 1));
  ((i   :== (λs. 0));
    WHILE (λs. (s sum) <= (s a)) DO
        ((  i   :== (λs. (s  i) + 1));
         ((tm   :== (λs. (s  tm) + 2));
          (sum  :== (λs. (s  tm) + (s sum)))))))
```

Verify this program using

1. the tactic-based method language

2. the structured ISAR language.

and compare the resulting proof scripts.

Hints:
- Use these given parenthesis's; the syntax setup of IMP is not really optimal this time !
- Do not forget to assume that the locations for i,tm, sum and a are pairwise distinct.
- Use update_def in the simplifier set to handle updates.

2.2 Exercise 44

Verify the IMP-program of the previous exercise *without* using the Hoare-calculus explicitly. The idea is to use the verification condition generator vc in theory `VC.thy` running over an annotated program, i.e. the program enriched by the crucial invariants.

(Here, we do not need an in-depth understanding of vc, we just apply it).

The abstract syntax of annotated programs is given in `VC` by the datatype:

```
datatype acom = Askip
              | Aass   loc aexp
              | Asemi  acom acom
              | Aif    bexp acom acom
              | Awhile bexp assn acom
```

(The assn in the Awhile-case is the invariant).

Note: The crucial theorem vc_sound allows for the reduction of the Hoare-triple

|− {pre} squareroot tm sum i a {post a i}

to a HOL-formula generated by vc.

Hints:
- Do not forget to assume that the locations for i,tm, sum and a are pairwise distinct.
- Give the annotated program aprog first (the let-statement may help here!).
- Prove the subgoal squareroot tm sum i a = astrip aprog, i.e. the annotated program must be the previously defined program squareroot if the annotations are "stripped away".
- Prove the subgoal pre = awp aprog (post a i), i.e. the weakest precondition computed from the program is equivalent to the precondition.

- apply theorem vc_sound.
- compute and solve the verification condition.
- Use update_def in the simplifier set to handle updates.