Advances in Programming Languages

APL6: Hoare logic

David Aspinall
(slides mostly by Ian Stark)

School of Informatics
The University of Edinburgh

Thursday 29 January 2009
Semester 2 Week 3

Topic: Some formal verification

The next three lectures will be about some techniques and tools for formal verification, specifically:

- Hoare logic
- JML: The Java Modeling Language
- ESC/Java 2: The Extended Static Checker for Java

Outline

1. Introduction

2. Axioms, meaning and truth

3. Applications

4. Closing

First-order logic

A formal language for describing certain kinds of logical assertion.

Variables $x, y, z, x_1, \ldots$ Terms $e ::= x \mid f(e_1, \ldots, e_n)$

$$\begin{aligned} \text{Formulae} \quad P, Q \; ::= \; & \mathbf{true} \mid \mathbf{false} \mid R(e_1, \ldots, e_n) \\ \mid \; & P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \neg P \\ \mid \; & \forall x.P \mid \exists x.P \end{aligned}$$

A *function* like $f(\ldots)$ has a fixed number of arguments, its *arity*. This might be zero, one or more. For example: 5, $sqrt(-)$, $+$.

A *predicate* like $R(\ldots)$ also has an arity: zero (a *proposition*), one (a *predicate*), or more (a *relation*). For example: $\mathbf{true}$, $Even(-)$, $<$, $=$, $Divides(-,-)$.

$$\forall x, y.(x > 5) \wedge (y > x) \rightarrow (x + y > 10)$$

A simple imperative language

Pick a minimal language of commands and variable assignment.

Variables a, b, i, n, . . .   Expressions E, B ::= a | F(E1,...,En)

Code C ::= skip | a:=E | C;C
| if B then C else C | while B do C

Variables like a, b here are storage cells, distinct from logical variables x, y.

Functions F have an arity, and we assume useful ones like 0, 1, +, sqrt(−).

For example, the following computes the factorial of n and places it in variable m:

```
i := n; a := 1; while i>0 do (a := a∗i; i: = i−1); m := a
```

Hoare triples

A *Hoare triple* is an assertion about the behaviour of a program fragment.

$$\{\mathrm{P}\}\ \mathsf{C}\ \{\mathrm{Q}\}$$

Here we have:

- An imperative program C.
- A *precondition* P and a *postcondition* Q: logical formulae concerning the state of the program variables.

The triple asserts that for any terminating run of the program, if P holds before then Q holds afterwards.

$$\{\mathsf{a} > 3\}\ \mathsf{b} := \mathsf{a}+\mathsf{a}\ \{\mathsf{b} > 6\}$$

$$\{\mathsf{d} > z \wedge \mathsf{d'} > z\}\ \mathsf{d} := \mathsf{d}*\mathsf{d'}\ \{\mathsf{d} > z^2\}$$

$$\{\mathbf{true}\}\ \mathsf{while\ i{>}0\ do\ i := i{-}1}\ \{\mathsf{i} \leqslant 0\}$$

Partial vs. Total

Partial: $\{P\}$ C $\{Q\}$ does not assert that C will terminate when started in a state satisfying P, only that Q will hold if it does.

The alternative *total* triple $[P]$ C $[Q]$ does assert that C terminates, but in practice methods for proving termination are often quite different to methods for proving properties like Q.

Hypothetical: $\{P\}$ C $\{Q\}$ makes no claim that P actually will be true when C is executed, only what will happen if it is.

Imprecise: $\{P\}$ C $\{Q\}$ may not include all that can be deduced about C.

For example, $\{\mathbf{true}\}$ C $\{\mathbf{true}\}$ is always valid, but rarely useful.

Hoare rules

Hoare set out a number of rules for how to deduce triples.

$$\frac{}{\{P\}\ \text{skip}\ \{P\}} \qquad \frac{\{P\}\ \text{C}\ \{Q\} \qquad \{Q\}\ \text{C'}\ \{R\}}{\{P\}\ \text{C;C'}\ \{R\}} \qquad \frac{}{\{P[\text{E/x}]\}\ \text{x:=E}\ \{P\}}$$

$$\frac{\{P \wedge (\text{B} = \text{true})\}\ \text{C}\ \{Q\} \qquad \{P \wedge (\text{B} \neq \text{true})\}\ \text{C'}\ \{Q\}}{\{P\}\ \text{if B then C else C'}\ \{Q\}}$$

$$\frac{\{P \wedge (\text{B} = \text{true})\}\ \text{C}\ \{P\}}{\{P\}\ \text{while B do C}\ \{P \wedge (\text{B} \neq \text{true})\}}$$

$$\frac{P \rightarrow P' \quad \{P'\}\ \text{C}\ \{Q'\} \quad Q' \rightarrow Q}{\{P\}\ \text{C}\ \{Q\}}$$

Rules have also been proposed for several other programming language features: concurrency, procedures, local variables, pointers,. . .

In fact, the last rule is not as strong as it might be, but this was not realised for several years.

See for example [Nipkow CSL 2002 §3] for some of the history.

Truth and beauty

We write $\vdash \{P\}\ \mathsf{C}\ \{Q\}$ when a triple can be derived using the rules. But is such a triple true? This depends on the meaning of C, its *semantics*. Which is what, exactly?

- Hoare proposed an *axiomatic semantics*: the derivable triples $\vdash \{P\}\ \mathsf{C}\ \{Q\}$ are what define the meaning of C.
- An alternative is to define the behaviour of C separately, and write $\vDash \{P\}\ \mathsf{C}\ \{Q\}$ when a triple holds true in this other semantics.

There are various such ways to define the behaviour of C:

- *Operational semantics*: how one term executes to give another.
- *Denotational semantics* maps programs into a mathematical domain.
- An *abstract machine* executes steps in a simplified processor.

In all cases we then want to compare $\vdash$ (derived) with $\vDash$ (observed).

Operational semantics

An operational semantics here must track commands C and program states S, where S(x) gives the value of variable x in state S.

- A *small-step* semantics S, C → S', C' reduces programs little by little:

$$\text{S, (a:=5;C)} \quad \longrightarrow \quad \text{S'[a} \leftarrow \text{5], C}$$

- A *big-step* semantics S, C ⇓ S' evaluates programs to a final state:

$$\text{S, (i:=5;j:=1;while i>0 do (i:=i}-\text{1;j:=j}*\text{2))} \Downarrow \text{S[i} \leftarrow \text{0,j} \leftarrow \text{32]}$$

Either of these can themselves be defined by derivation rules, using the approach of *Structural Operational Semantics*. [Plotkin 1981]

Soundness and completeness

Given a semantics, we can identify which triples are valid:

$$\vDash \{P\}\ C\ \{Q\} \quad \overset{\text{def}}{\iff} \quad \forall S,T\ .\ (P(S) \land S,C \Downarrow T) \rightarrow Q(T)$$

This gives a means to assess the derivation rules for triples:

Soundness Every derivable triple is valid:

$$\vdash \{P\}\ C\ \{Q\} \quad \Longrightarrow \quad \vDash \{P\}\ C\ \{Q\}$$

Completeness Every valid triple can be derived using the rules:

$$\vDash \{P\}\ C\ \{Q\} \quad \Longrightarrow \quad \vdash \{P\}\ C\ \{Q\}$$

Reasoning and specification

Hoare logic supports quite general reasoning about imperative programs and their behaviour. However, the two most common applications are:

Specification Stating what properties a program ought to have, either by annotating existing code, or before any is written.

Verification Checking that a program does indeed have these desired properties.

In practice, this means generalising pre- and postconditions to include:

Assertions about the state at some point within a program.

Loop invariants to hold at each repeat of a loop.

Object invariants that each method is to maintain.

Method constraints as pre- and postconditions on method invocation.

Hoare in verification tools

The general approach for Hoare-style formal verification tools is this:

- A programmer annotates source code, or a library interface.
- A tool analyses the code and attempts to show that all the assertions given can be derived using the standard rules.
- The tool may be able to do this unassisted.
- If not, it emits *verification conditions*, purely logical assertions that need to be checked.
- These may be passed on to an automated theorem prover, or some other *decision procedure*.
- In extreme cases verification conditions may not be solved automatically and require interactive theorem proving by an expert or the provision of extra hints.

Design by Contract^{TM}

Design by Contract^{TM} (DBC) is a software design methodology promoted by Bertrand Meyer and the *Eiffel* programming language.

DBC makes Hoare logic a vital component in program development, strengthening it to the notion of a *contract*:

- The precondition of a procedure imposes an obligation on any caller;
- In return, the procedure must guarantee that the specified postcondition will hold when it exits.

The contract also includes additional information such as side-effects, invariants, and error conditions.

NB: this modifies the *hypothetical* aspect of Hoare logic, where a precondition is "supposing"

Lightweight Verification

Proving (and writing) arbitrary assertions can be arbitrarily difficult.

In *lightweight verification* things are simplified by focusing on standard properties of common interest rather, than full functional correctness.

Exception freedom no uncaught exception is raised.

Arithmetic safety no arithmetic expression divides by zero or overflows.

Race freedom access to shared state does not conflict in different threads.

Standard properties are easy for the programmer to write, providing shorthands for possibly complex logical expressions.

Standard properties can be easier for tools to handle, using ad hoc static analyses or decidable fragments of logic.

If a tool cannot establish a property, the programmer may be able to add additional annotations, or may have to rewrite.

Homework

The next lectures will be on JML and ESC/Java 2, two tools that apply Hoare Logic and DBC. Before Monday, read the following two short articles:

- Leavens and Cheon. Design by Contract with JML.
- Burdy et al. An overview of JML tools and applications.

Both available from http://jmlspecs.org

Extra challenge activity: install and run JML or ESC/Java 2.
(Hints: Java 1.6, Eclipse 3.4, suitable Eclipse plugins)

Summary

- Hoare logic triples $\{P\}$ C $\{Q\}$ make logical assertions about imperative code.
- The *soundness* and *completeness* of Hoare reasoning can be tested with respect to a program's *semantics*.
- Hoare assertions are used in *specification* to annotate programs and libraries.
- Tools can carry out automated *verification* against these assertions.
- Design by Contract$^{\text{TM}}$ strengthens these into *contracts*.
- In *lightweight* verification, the focus is on standard "goodness" properties expressed succinctly.