Advances in Programming Languages

Lecture 12: Specification and Verification

Ian Stark

School of Informatics
The University of Edinburgh

Friday 4 November 2016
Semester 1 Week 7

Topic: Augmented Languages for Correctness and Certification

The next block of lectures cover some language techniques and tools for improving program correctness:

- Specification and Verification
- Practical Java tools for Correctness
- Augmented Programming and Certifying Correctness

The focus here is not necessarily on changing what a program does, or making it do that thing faster, or using less memory, or less power. Instead we want to make sure that what it does is the right thing to do.

This lecture introduces *Hoare Logic*, a classic language for describing and checking the properties of programs.

Also known as *Floyd-Hoare Logic*

Exam Dates !

The University of Edinburgh holds examinations in three *diets* during the year:

December Diet at the end of Semester 1;
April/May Diet at the end of the academic year;
August Diet principally for resit examinations.

There are APL exams in two of these diets this year. If you are

- Studying for an undergraduate or postgraduate degree at the University of Edinburgh
- Visiting from another university for the year

then you must sit the APL exam in **April/May**. The exam timetable will be published on Monday 6 March 2017.

If you are

- Visiting from another university for this semester only

then you must sit the APL exam in **December**. The exam timetable has been published, and can be checked online.

Previous Homework

CPU features that make concurrency even trickier

Find out the meaning of the following words, in the context of CPU architecture and execution.

- Pipelining
- Pipeline hazard
- Superscalar
- Out-of-order execution
- Speculative execution
- Branch prediction

First-order logic

First-order logic is a formal language for describing certain kinds of logical assertion.

Syntax of first-order logic

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

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

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: $\mathbf{true}$, $\mathrm{Even}(-)$, $<$, $=$, $\mathrm{Divides}(-,-)$.

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

A simple imperative language

Pick a minimal language of commands and variable assignment.

Syntax of a small while-language

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 formulas 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 if it does then Q will hold.

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.

This also means that many different triples may hold for a piece of code C

$$\{P\}\ \mathsf{C}\ \{Q\}, \qquad \{P'\}\ \mathsf{C}\ \{Q'\}, \qquad \{P''\}\ \mathsf{C}\ \{Q''\} \quad \ldots$$

without necessarily any relation among the P, P' P'' and Q, Q', Q''.

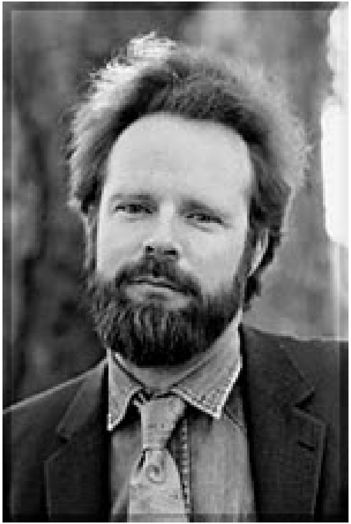
Creators of Floyd-Hoare logic

Inventor of:

- Floyd-Warshall shortest-path algorithm for graphs
- Floyd-Steinberg dithering algorithm for images
- Floyd-Hoare logic for programs
- ...

1978 ACM Turing Award

"For having a clear influence on methodologies for the creation of efficient and reliable software, and for helping to found the following important subfields of computer science: the theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, and analysis of algorithms"



Robert W 'Bob' Floyd

Creators of Floyd-Hoare logic

Sir Charles Antony Richard 'Tony' Hoare

Inventor of:

- Quicksort
- Communicating Sequential Processes (CSP)
- Monitors (locks & condition variables)
- Hoare Logic
- Null pointers
- ...

1980 ACM Turing Award

"For his fundamental contributions to the definition and design of programming languages"

Hoare rules

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

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

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

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

$$\frac{P \to P' \qquad \{P'\}\ \mathsf{C}\ \{Q'\} \qquad Q' \to Q}{\{P\}\ \mathsf{C}\ \{Q\}}$$

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 many 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?

> Beauty is truth, truth beauty — that is all
> Ye know on earth, and all ye need to know
>
> Keats, *Ode on a Grecian Urn*

Keats didn't think a Man of Achievement should fuss about finding truth, instead praising what he called *Negative Capability*:

> . . . , that is, when a man is capable of being in uncertainties, mysteries, doubts, without any irritable reaching after fact and reason.

So he probably wouldn't think much of the talk about now.

Truth and beauty

We write $\vdash \{P\}\ C\ \{Q\}$ 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 $\vdash \{P\}\ C\ \{Q\}$ are what define the meaning of C.
- An alternative is to define the behaviour of C separately, and write $\vDash \{P\}\ C\ \{Q\}$ 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 $\mathsf{S},\ \mathsf{C} \rightarrow \mathsf{S'},\ \mathsf{C'}$ reduces programs little by little:

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

- A *big-step* semantics $\mathsf{S},\ \mathsf{C} \Downarrow \mathsf{S'}$ evaluates programs to a final state:

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

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*:

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

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

Soundness Every derivable triple is valid:

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

Completeness Every valid triple can be derived using the rules:

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

Gödel's celebrated theorem tells us we can only hope for *relative* completeness in useful logics.

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 as follows.

- 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$^{\text{TM}}$

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.

Pointer validity no null pointer is ever dereferenced.

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 shorthand 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 the code.

Homework

The lecture on Tuesday will look at some tools for checking Java programs, including those that apply Hoare Logic and Design by Contract™. Before then:

1. Read this

Gary Leavens and Yoonsik Cheon
Design by Contract with JML
http://www.jmlspecs.org/jmldbc.pdf:

2. Do this

- Find some information online about *assertions* in Java — a tutorial, Q+A, a discussion, a blog post, . . .
- Post a link to the Piazza group, with any comments you have on what you thought helpful or otherwise.

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™ strengthens these into *contracts*.
- In *lightweight verification*, the focus is on standard good properties: expressed succinctly and widely understood.