6894 · Lightweight Formal Methods

Lecture 14 : Hoare Logic

Daniel Jackson, April 11, 2005

Code verification & checking

Program verification has been a research topic for 35 years! Practical applications have only just begun to appear.

Why?

- Moore's Law: need lots of computational resources;
- Until recently, mostly pursued on algorithms, for toy languages; application to languages with encapsulation, objects, inheritance, and to large code bases is a new interest;
- Formal methods were for a long time of interest only to theoreticians; attitudes are changing and more people are interested in correctness and dependability now — including the kind of researchers who can build tools.

Main approaches:

Model checking

extract control flow skeleton, check for deadlock, etc, or in sequential code, for violations of API usage rules

Examples

SLAM (Microsoft)
Blast (Berkeley)
Java Pathfinder (~~Jet Prop~~ NASA)
Bandera (Kansas State)

Dataflow analysis

Check API usage rules, null pointers, array bounds violations, etc.

Examples

PreFix, PreFAST (Microsoft)
Saturn (Stanford & - Aiken)
Metal (Stanford - Engler)
TVLA (Tel Aviv)

Ad hoc analyses

Do pattern matching to look for standard symptoms of bugs and poor style.

Examples.

FindBugs. (UMD)
Splint (UMD)

Type systems.

Enrich a type system to account for null pointers, what vars are readable / writeable,

Examples.

Vault (Microsoft)
Cqual (Berkeley)
Cyclone (Harvard)

These are mostly designed to be safe systems programming languages. So they don't offer much

beyond a conventional strongly typed language like Java for the purposes of checking.

Program verification.

check code against basic assertions, array bounds violations, null pointer derefs, etc.,
using classical verification technology.

Examples.

ESC/Java (Compaq Systems Research Center)

Why learn about program verification?

- Classic ideas every computer scientist should know;
- An old technology becoming new again;
- Potential to address complex data (where the other approaches tend to work only for control flow);
- Understanding the ideas can help you become a better programmer.

Lightweight PV - an oxymoron?

Few approaches to programming that have been proposed have been as heavyweight as program verification! Lots of interest now in making it lightweight and cost-effective.

Topics:

- Hoare Logic
  - how to reason compositionally about code
  - pre- and post-conditions
  - loop invariants

- Dijkstra's weakest preconditions
  - goal-directed reasoning
  - the approach used in most tools that do PV.

- Refinement
  - reasoning about levels of abstraction
  - abstract data types.

Hoare Logic.

'An Axiomatic Basis For Computer Programming'
Hoare, CACM 1969.

- one of the most widely cited papers in CS.
- laid the basis for whole field of formal verification

Bob Floyd had earlier devised a way of reasoning about programs (as flowcharts) using assertions relating the state at some point to the initial state.

Tony Hoare's contribution was to develop a compositional approach: reasoning about the correctness of the whole from the correctness of its parts.

The logics we've seen so far for reasoning about dynamic behaviour (CTL and LTL) are ~~ENDO~~EXOGENOUS: you write a logical formula about some system expressed in another formalism (eg. Kripke structures).

Hoare Logic, in contrast is ENDOGENOUS: the component and its property are together in the same formula.

Another important difference: CTL and LTL are used mainly in model checking for SEMANTIC reasoning; Hoare logic is SYNTACT

SYNTACTIC reasoning:

program verification entirely by symbol manipulation.

For a long time, researchers have talked about CALCULATING programs but it's still not possible except for small programs.

Hoare Triples

$\{P\}\ S\ \{Q\}$

is a formula that's true iff
program statement (or component) S
when executed in a state satisfying P
either terminates in a state satisfying Q
or doesn't terminate.

This is called a partial correctness assertion:
if S terminates, it terminates in a state satisfying Q.
A total correctness formula would say
S does terminate, in a state satisfying Q.

P is called the precondition
Q is called the post condition.

In Hoare's original formulation, a triple was written
$P\ \{S\}\ Q$
but now it's always written with the $\{\}$ around the conditions, to make them look like comments.

Note $\{false\}\ S\ \{Q\}$
$\{P\}$ while (true) do skip; $\{Q\}$
are both valid, irrespective of P, S, Q.

Axioms.

SKIP : $\{P\}\ skip\ \{P\}$

ASSIGN : $\{P[u := t]\}\ u := t\ \{P\}$

Rules of inference:

COMPOSITION: $$\frac{\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1 ; S_2\ \{Q\}}$$

CONDITIONAL: $$\frac{\{P \wedge B\}\ S_1\ \{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

LOOP: $$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S\ \{P \wedge \neg B\}}$$

CONSEQUENCE: $$\frac{P \Rightarrow P_1,\ \{P_1\}\ S\ \{Q_1\},\ Q_1 \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Examples:

$\{x = y\}\ x := x+1;\ y := y+1;\ \{x = y\}$

By assignment axiom:

$$\{x = y+1\}\ y := y+1\ \{x = y\}$$

By assignment axiom again.

$$\{x+1 = y+1\}\ x := x+1\ \{x = y+1\}$$

Applying composition rule:

$$\{x+1 = y+1\}\ x := x+1;\ y := y+1\ \{x = y\}$$

and since $x = y \Rightarrow x+1 = y+1$

using the consequence rule we have

$$\{x = y\}\ x := x+1;\ y := y+1\ \{x = y\}$$

Note that post condition can't mention variables from the prestate. So how do we say, eg, that

$$x := x+1 \;;\; x := x+1$$

increases x by 2?

Common trick is to use <u>auxiliary variables</u>.

$$\{x = X\} \quad x := x+1 \;;\; x := x+1; \; \{x = X+2\}$$

Example.

Show that SWAP ≜ $t := x \;;\; x := y \;;\; y := t$ swaps the values of x and y.

Triple is $\{x = X \wedge y = Y\}$ SWAP $\{x = Y \wedge y = X\}$

$$\{x = Y \wedge t = X\} \; y := t \; \{x = Y \wedge y = X\} \qquad \text{(ASSIGN)}$$
$$\{y = Y \wedge t = X\} \; x := y \; \{x = Y \wedge t = X\} \qquad \text{(ASSIGN)}$$
$$\{y = Y \wedge x = X\} \; t := x \; \{y = Y \wedge t = X\} \qquad \text{(ASSIGN)}$$

Then apply composition (twice) and consequence.

These proofs are very tedious, especially because of all the applications of the composition & consequence rules.

So instead, present as proof outlines.

$\{x = X \wedge y = Y\}$
$\{y = Y \wedge x = X\}$
$t := x;$
$\{y = Y \wedge t = X\}$
$x := y;$
$\{x = Y \wedge t = X\}$
$y := t;$
$\{x = Y \wedge y = X\}$

Note:

a proof outline is a program with conditions written as assertions!

Can formalize a system for generating proof outlines:

$$\frac{\{P\}\ S_1^*\ \{R\},\ \{R\}\ S_2^*\ \{Q\}}{\{P\}\ S_1^*\ \{R\}\ S_2^*\ \{Q\}}$$

↑ intermediate assertion stays!

S_i^* is program
S_i interspersed with annotations.

```
{true}
{1 = 0!}                                   consq
y=1;
{y = 0!}                                   assign
z=0;
{y = z!}                                   assign
while (z != x) {
  {y = z! ∧ z≠x}
  {y·(z+1) = (z+1)!}                       consq
  z = z+1;
  {y·z = z!}                               assign
  y = y*z;
  {y = z!}                                 assign
}
{y = z! ∧ ¬(z≠x)}                          while
{y = x!}                                   consq
```

so {true} FACT {y = x!}

Why isn't $x \geq 0$ needed as a precondition?

Note the role of the LOOP INVARIANT {y = z!}
(It's the {P} in the WHILE rule)

Replacing a constant by a variable

Summing the elements of an array

$\{n \geq 0\}$ SUM $\{s = \Sigma j : 0 \leq j < n : b[j]\}$

```
i, s := 0, 0 ;
while i≠n {
    s := s + b[i] ;
    i := i + 1 ;
    }
```

invariant :

$0 \leq i \leq n \;\wedge\; s = \Sigma j : 0 \leq j < i : b[j]$

(i: var for const)

Examples of loop invariants (from Gries)

Deleting a conjunct

Problem: find approx root:

$$\{n \geq 0\} \text{ ROOT } \{0 \leq a^2 \leq n < (a+1)^2\}$$

a is the largest integer that is at most $\sqrt{n}$

```
a := 0
while (a+1)^2 ≤ n
    a := a+1;
```

invariant: $0 \leq a^2 \leq n$

final postcond: $0 \leq a^2 \leq n \wedge n < (a+1)^2$

note:

negation of deleted conjunct is the looping condition.

ensures automatically that $inv \wedge \neg$ loop-cond $\Rightarrow$ post.

Loop invariant

is an induction hypothesis.

must be an invariant: preserved by loop body.

strong enough to imply the postcondition

weak enough to be established by precondition & the loop

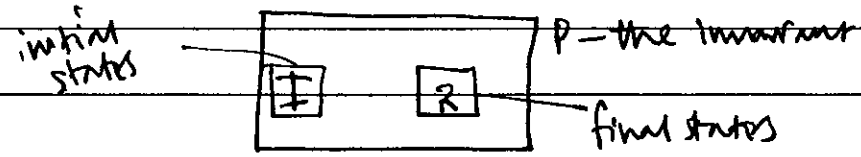
Key idea in finding a loop invariant:

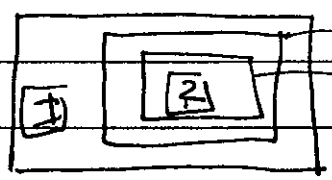
it expresses progress through the series of iterations.

it's a generalized form of the postcondition.

David Gries:

think of finding the invariant by 'inflating a balloon':





each iteration 'deflates the balloon'

So to obtain the invariant

start from the postcondition required R

and systematically weaken it:

delete a conjunct $A \wedge B \wedge C \rightsquigarrow A \wedge C$

replace const by variable $x \leq b[1:10] \rightsquigarrow x \leq b[1:i] \wedge 1 \leq i$

enlarge range of variable $5 \leq i \leq 10 \rightsquigarrow 0 \leq i \leq 10$

and ∧ disjunct $A \rightsquigarrow A \vee B$

NOT useful as a guide for finding invs.

Enlarging the range of a variable

Linear Search $\{\exists j.\ x = b[j]\}$ FIND $\{i = \min\{j \mid x = b[j]\}$

```
i := 0 ;
while b[i] ≠ x
    i := i + 1
```

invariant is:

$$0 \le i \le \min\{j \mid x = b[j]\}$$

Can also be viewed as an example of deleting a conjunct. Where spec is written with post condition.

$$\left[\forall j:\ 0 \le j < i\ .\ x \ne b[j]\right] \ \wedge\ x = b[i]$$

Problems with Hoare Logic

1. Capturing complexities of programming languages

In the presence of aliasing, the assignment rule is not valid. But no aliasing in languages like Java: parameter passing is by value.

Semantics of integer operations. An advantage of HL: see the classic paper. Can express different integer overflow semantics axiomatically.

Encapsulation, inheritance, etc. Those are the most serious problems. Lots of recent research on this by Rustan Leino et al.

2. To use the logic, you need to go backwards because of the ASSIGN rule, and yet the logic itself has no directionality. Hence Dijkstra's weakest precondition calculus.

3. Finding loop invariants: can't in general be automated and requires considerable insight.

4. Checking implications (for the CONSEQUENCE rule): undecidable when you have any non-trivial datatypes. A major stumbling block for automation.