Introduction, Classical Hoare Logic

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

12 décembre 2012

Preliminaries

- Very first question: lectures in English or in French?

Preliminaries

- Very first question: lectures in English or in French?
- Lectures 1,2,7,8: Claude Marché
- Lectures 3,4,5,6: Guillaume Melquiond
- On some week: practical lab, support for project
- Evaluation:
  - final exam E in March 2013
  - project P, using the Why3 tool (`http://why3.lri.fr`)
  - final mark = $(2E + P + max(E, P))/4$
- internships (stages)
- Warning: room change in January
- Slides and lectures notes on the web page
  `http://www.lri.fr/~marche/MPRI-2-36-1/`

Outline

General Objectives

Ultimate Goal

Verify that software is free of bugs

Famous software failures:
`http://www.cs.tau.ac.il/~nachumd/horror.html`

This lecture

Computer-assisted approaches for verifying that
a software conforms to a specification

Some general approaches to Verification

Static analysis

- *model checking* (automata-based models)
- *abstract interpretation* (domain-specific model, e.g. numerical)
- verification: fully automatic dedicated algorithms

Deductive verification

- formal models using expressive logics
- verification = computer-assisted mathematical proof

Some general approaches to Verification

Refinement

- Formal models
- Code derived from model, correct by construction

A long time before success

Computer-assisted verification is an old idea

- Turing, 1948
- Floyd-Hoare logic, 1969

Success in practice: only from the mid-1990s

- Importance of the *increase of performance of computers*

A first success story:

- Paris metro line 14, using *Atelier B* (1998, refinement approach)
  `http://www.methode-b.com/documentation_b/ClearSy-Industrial_Use_of_B.pdf`

Other Famous Success Stories

- Flight control software of A380: *Astree* verifies absence of run-time errors (2005, abstract interpretation)
  `http://www.astree.ens.fr/`
- Microsoft's hypervisor: using Microsoft's *VCC* and the *Z3* automated prover (2008, deductive verification)
  `http://research.microsoft.com/en-us/projects/vcc/`
- Certified C compiler, developed using the *Coq* proof assistant (2009, correct-by-construction code generated by a proof assistant)
  `http://compcert.inria.fr/`
- L4.verified micro-kernel, using tools on top of *Isabelle/HOL* proof assistant (2010, Haskell prototype, C code, proof assistant)
  `http://www.ertos.nicta.com.au/research/l4.verified/`

Outline

Syntax: expressions

$$
\begin{array}{rcll}
e & ::= & n & \text{(integer constants)} \\
 & | & x & \text{(variables)} \\
 & | & e \ op \ e & \text{(binary operations)} \\
op & ::= & + \mid - \mid * & \\
 & | & = \mid \neq \mid < \mid > \mid \leq \mid \geq & \\
 & | & and \mid or &
\end{array}
$$

- Only one data type: unbounded integers
- Comparisons return an integer: 0 for “false”, -1 for “true”
- There is no division

Consequences:

- Expressions are always well-typed
- Expressions always evaluate without error
- Expressions do not have any side effect

Syntax: statements

s	::=	`skip`	(no effect)
	|	x := e	(assignment)
	|	s; s	(sequence)
	|	`if` e `then` s `else` s	(conditional)
	|	`while` e `do` s	(loop)

- Condition in **`if`** and **`while`**: 0 is “false”, non-zero is “true”
- **`if`** without **`else`**: syntactic sugar for **`else`** `skip`.

Consequences:

- Statements have side effects
- All programs are well-typed
- There is no possible *runtime error*: all programs execute until their end or infinitely

Running Example

Three global variables `n`, `count`, and `sum`

```
count := 0; sum := 1;
while sum ≤ n do
  count := count + 1; sum := sum + 2 * count + 1
```

Running Example

Three global variables `n`, `count`, and `sum`

```
count := 0; sum := 1;
while sum ≤ n do
  count := count + 1; sum := sum + 2 * count + 1
```

Informal specification:

- at the end of execution of this program, `count` contains the square root of `n`, rounded downward
- e.g. for `n=42`, the final value of `count` is 6.

Operational semantics

[Plotkin 1981, structural operational semantics (SOS)]

- we use a standard *small-step semantics*
- *program state*: describes content of global variables at a given time. It is a finite map Σ associating to each variable x its current value denoted $\Sigma(x)$.
- Value of an expression e in some state Σ:
  - denoted $[\![e]\!]_\Sigma$
  - always defined, by the following recursive equations:

$$\begin{array}{rcl} [\![n]\!]_\Sigma & = & n \\ [\![x]\!]_\Sigma & = & \Sigma(x) \\ [\![e_1\ op\ e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma\ [\![op]\!]\ [\![e_2]\!]_\Sigma \end{array}$$

  - $[\![op]\!]$ natural semantic of operator op on integers (with relational operators returning 0 for false and -1 for true).

Semantics of statements

Semantics of statements: defined by judgment

$$\Sigma, s \quad \rightsquigarrow \quad \Sigma', s'$$

meaning: in state Σ, executing one step of statement s leads to the state Σ' and the remaining statement to execute is s'.
The semantics is defined by the following rules.

$$\frac{}{\Sigma, x := e \rightsquigarrow \Sigma\{x \leftarrow [\![e]\!]_\Sigma\}, \texttt{skip}}$$

$$\frac{\Sigma, s_1 \rightsquigarrow \Sigma', s_1'}{\Sigma, (s_1; s_2) \rightsquigarrow \Sigma', (s_1'; s_2)}$$

$$\frac{}{\Sigma, (\texttt{skip}; s) \rightsquigarrow \Sigma, s}$$

Semantics of statements, continued

$$\frac{[\![e]\!]_\Sigma \neq 0}{\Sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightsquigarrow \Sigma, s_1}$$

$$\frac{[\![e]\!]_\Sigma = 0}{\Sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightsquigarrow \Sigma, s_2}$$

$$\frac{[\![e]\!]_\Sigma \neq 0}{\Sigma, \texttt{while } e \texttt{ do } s \rightsquigarrow \Sigma, (s; \texttt{while } e \texttt{ do } s)}$$

$$\frac{[\![e]\!]_\Sigma = 0}{\Sigma, \texttt{while } e \texttt{ do } s \rightsquigarrow \Sigma, \texttt{skip}}$$

Execution of programs

- $\rightsquigarrow$: a binary relation over pairs (state,statement)
- transitive closure : $\rightsquigarrow^+$
- reflexive-transitive closure : $\rightsquigarrow^*$

In other words:

$$\Sigma, s \rightsquigarrow^* \Sigma', s'$$

means that statement s, in state Σ, reaches state Σ' with remaining statement s' after executing some finite number of steps.

Running example:

$$\{n = 42, count = ?, sum = ?\}, ISQRT \rightsquigarrow^* \{n = 42, count = 6, sum = 49\}, \texttt{skip}$$

Execution and termination

- any statement except `skip` can execute in any state
- the statement `skip` alone represents the final step of execution of a program
- there is no possible *runtime error*.

Definition

Execution of statement s in state Σ *terminates* if there is a state Σ' such that $\Sigma, s \rightsquigarrow^* \Sigma', \texttt{skip}$

- since there are no possible runtime errors, s does not terminate means that s *diverges* (i.e. executes infinitely).

Sequence lemma

Lemma (Sequence execution)

If

$$\Sigma, (s_1; s_2) \quad \rightsquigarrow^* \quad \Sigma', \mathit{skip}$$

then there exists an intermediate state Σ'' *such that*

$$\begin{aligned} \Sigma, s_1 \quad &\rightsquigarrow^* \quad \Sigma'', \mathit{skip} \\ \Sigma'', s_2 \quad &\rightsquigarrow^* \quad \Sigma', \mathit{skip} \end{aligned}$$

Proof.

Induction on the number of steps of execution. □

Outline

Propositions about programs

- To formally express properties of programs, we need a *formal specification language*
- We use *standard first-order logic*
- syntax of formulas:

$$p \quad ::= \quad e \mid p \wedge p \mid p \vee p \mid \neg p \mid p \Rightarrow p \mid \forall v,\ p \mid \exists v,\ p$$

- v : *logical variable* identifiers
- e : program expressions, augmented with logical variables

Semantics of formulas

$[\![p]\!]_\Sigma$:

- semantics of formula p in program state Σ
- is a logic formula where no program variables appear anymore
- defined recursively as follows.

$$\begin{array}{rcl} [\![e]\!]_\Sigma & = & [\![e]\!]_\Sigma \neq 0 \\ [\![p_1 \wedge p_2]\!]_\Sigma & = & [\![p_1]\!]_\Sigma \wedge [\![p_2]\!]_\Sigma \\ & \vdots & \end{array}$$

where semantics of expressions is augmented with

$$\begin{array}{rcl} [\![v]\!]_\Sigma & = & v \\ [\![x]\!]_\Sigma & = & \Sigma(x) \end{array}$$

Notations:

- $\Sigma \models p$: the formula $[\![p]\!]_\Sigma$ is *valid*
- $\models p$: formula $[\![p]\!]_\Sigma$ holds in all states Σ.

Hoare triples

- *Hoare triple* : notation $\{P\}s\{Q\}$
- P : formula called the *precondition*
- Q : formula called the *postcondition*

Definition (Partial correctness)

Hoare triple $\{P\}s\{Q\}$ is said *valid* if:
for any states Σ, Σ', if

- $\Sigma, s \rightsquigarrow^* \Sigma', \texttt{skip}$ and
- $\Sigma \models P$

then $\Sigma' \models Q$

In other words: if s is executed in a state satisfying its precondition, then *if it terminates*, the resulting state satisfies its postcondition

Examples

Examples of valid triples for partial correctness:

- $\{x = 1\}x := x + 2\{x = 3\}$
- $\{x = y\}x := x + y\{x = 2 * y\}$
- $\{\exists v,\ x = 4 * v\}x := x + 42\{\exists w,\ x = 2 * w\}$
- $\{true\}\texttt{while 1 do skip}\{false\}$

Our running example:

$\{n \geq 0\}ISQRT\{count*count \leq n \wedge n < (count+1)*(count+1)\}$

Hoare logic

Set of *inference rules* producing triples

$$\frac{}{\{P\}\texttt{skip}\{P\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x := e\{P\}}$$

$$\frac{\{P\}s_1\{Q\} \qquad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

- Notation $P[x \leftarrow e]$: replace all occurrences of program variable x by e in P.

Hoare Logic, continued

Consequence rule:

$$\frac{\{P'\}s\{Q'\} \qquad \models P \Rightarrow P' \qquad \models Q' \Rightarrow Q}{\{P\}s\{Q\}}$$

- Example: proof of

$$\{x = 1\}x := x + 2\{x = 3\}$$

Hoare Logic, continued

Rules for if and while :

$$\frac{\{P \wedge e \neq 0\} s_1 \{Q\} \qquad \{P \wedge e = 0\} s_2 \{Q\}}{\{P\}\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2\{Q\}}$$

$$\frac{\{I \wedge e \neq 0\} s \{I\}}{\{I\}\texttt{while } e \texttt{ do } s\{I \wedge e = 0\}}$$

- I is called a *loop invariant*.

Soundness

Theorem (Soundness of Hoare logic)

This set of rules is correct: any derivable triple is valid.

This is *proved by induction on the derivation tree* of the considered triple.
For each rule: assuming that the triples in premises are valid, we show that the triple in conclusion is valid too.

Example: isqrt(42)

Exercise: prove of the triple

$$\{n \geq 0\} ISQRT \{count * count \leq n \wedge n < (count + 1) * (count + 1)\}$$

Example: isqrt(42)

Exercise: prove of the triple

$$\{n \geq 0\} ISQRT \{count * count \leq n \wedge n < (count + 1) * (count + 1)\}$$

Warning

Finding an adequate loop invariant is a major difficulty

Completeness

Two major difficulties for proving a program

- *guess the appropriate intermediate formulas* (for sequence, for the loop invariant)
- *prove the logical premises of consequence rule*

Theoretical question: completeness. Are all valid triples derivable from the rules?

Theorem (Relative Completeness of Hoare logic)

The set of rules of Hoare logic is relatively complete: if the logic language is expressive enough, then any valid triple $\{P\}s\{Q\}$ can be derived using the rules.

[Cook, 1978]

"Expressive enough" is for example Peano arithmetic (non-linear integer arithmetic)

Gives only hints on how to effectively determine suitable loop invariants (see the theory of abstract interpretation *[Cousot, 1990]*)

Outline

Annotated Programs

Goal

Add automation to the Hoare logic approach

We augment our simple language with *explicit loop invariants*

s	$::=$	`skip`	(no effect)
	$\mid$	$x := e$	(assignment)
	$\mid$	$s; s$	(sequence)
	$\mid$	`if` e `then` s `else` s	(conditional)
	$\mid$	`while` e `invariant` I `do` s	(annotated loop)

- The operational semantics is unchanged.

Weakest liberal precondition

[Dijkstra 1975]

Function $\mathrm{WLP}(s, Q)$:

- s is a statement
- Q is a formula
- returns a formula

It should return the *minimal precondition* P that validates the triple $\{P\}s\{Q\}$

Definition of $\mathrm{WLP}(s, Q)$

Recursive definition:

$$
\begin{array}{rcl}
\mathrm{WLP}(\texttt{skip}, Q) & = & Q \\
\mathrm{WLP}(x := e, Q) & = & Q[x \leftarrow e] \\
\mathrm{WLP}(s_1; s_2, Q) & = & \mathrm{WLP}(s_1, \mathrm{WLP}(s_2, Q)) \\
\mathrm{WLP}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q) & = & \\
(e \neq 0 \Rightarrow \mathrm{WLP}(s_1, Q)) & \wedge & (e = 0 \Rightarrow \mathrm{WLP}(s_2, Q))
\end{array}
$$

Definition of $\mathrm{WLP}(s, Q)$, continued

$$
\begin{array}{lll}
\mathrm{WLP}(\texttt{while } e \texttt{ invariant } I \texttt{ do } s, Q) = & & \\
\quad I \wedge & & \text{(invariant true initially)} \\
\quad \forall v_1, \ldots, v_k, & & \\
\qquad (((e \neq 0 \wedge I) \Rightarrow \mathrm{WLP}(s, I)) & & \text{(invariant preserved)} \\
\qquad \wedge((e = 0 \wedge I) \Rightarrow Q))[w_i \leftarrow v_i] & & \text{(invariant implies post)}
\end{array}
$$

where $w_1, \ldots, w_k$ is the set of assigned variables in statement s and $v_1, \ldots, v_k$ are fresh logic variables

Examples

$$\mathrm{WLP}(x := x + y, x = 2y) \;\equiv\; x + y = 2y$$

Examples

$$\mathrm{WLP}(x := x + y, x = 2y) \;\equiv\; x + y = 2y$$

$$\mathrm{WLP}(\texttt{while } y > 0 \texttt{ invariant } \mathit{even}(y) \texttt{ do } y := y - 2, \mathit{even}(y)) \;\equiv$$

Examples

$$\mathrm{WLP}(x := x + y, x = 2y) \;\equiv\; x + y = 2y$$

$$\begin{aligned}\mathrm{WLP}(&\texttt{while } y > 0 \texttt{ invariant } even(y) \texttt{ do } y := y - 2, even(y)) \;\equiv \\ & even(y) \wedge \\ & \forall v, ((v > 0 \wedge even(v)) \Rightarrow even(v - 2)) \\ & \quad \wedge ((v \leq 0 \wedge even(v)) \Rightarrow even(v))\end{aligned}$$

Soundness

Theorem (Soundness)

For all statement s and formula Q, $\{\mathrm{WLP}(s, Q)\}s\{Q\}$ is valid.

Proof by induction on the structure of statement s.

Consequence

For proving that a triple $\{P\}s\{Q\}$ is valid, it suffices to prove the formula $P \Rightarrow \mathrm{WLP}(s, Q)$.

Application

Demo with the *Why3* tool (`http://why3.lri.fr/`)

See file `imp_isqrt.mlw`

(This is the tool to use for the project. version 0.80)

Minimality

Theorem (Weakest precondition property)

For any triple $\{P\}s\{Q\}$ *that is derivable, we have* $\models P \Rightarrow \mathrm{WLP}(s, Q)$.

Outline

Exercise 1

Consider the following (inefficient) program for computing the sum $a + b$.

```
x := a; y := b;
while y > 0 do
   x := x + 1; y := y - 1
```

- Propose a post-condition stating that the final value of x is the sum of the values of a and b
- Find an appropriate loop invariant
- Prove the program.

Exercise 2

The following program is one of the original examples of Floyd.

```
q := 0; r := x;
while r ≥ y do
    r := r - y; q := q + 1
```

- Propose a formal precondition to express that x is assumed non-negative, y is assumed positive, and a formal post-condition expressing that q and r are respectively the quotient and the remainder of the Euclidean division of x by y.
- Find appropriate loop invariant and prove the correctness of the program.

Exercise 3

Let's assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of x by 2 and its remainder. The following program is supposed to compute, in variable r, the power x^n.

```
r := 1; p := x; e := n;
while e > 0 do
  if mod2(e) ≠ 0 then r := r * p;
  p := p * p;
  e := div2(e);
```

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.

Exercise 4

The Fibonacci sequence is defined recursively by $fib(0) = 0$, $fib(1) = 1$ and $fib(n+2) = fib(n+1) + fib(n)$. The following program is supposed to compute *fib* in linear time, the result being stored in y.

```
y := 0; x := 1; i := 0;
while i < n do
   aux := y; y := x; x := x + aux; i := i + 1
```

- Assuming *fib* exists in the logic, specify appropriate pre- and post-conditions.
- Prove the program.

Exercise (Exam 2011-2012)

In this exercise, we consider the simple language of the first lecture of this course, where expressions do not have any side effect.

1. *Prove that the triple*

$$\{P\}x := e\{\exists v,\ e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}$$

   is valid with respect to the operational semantics.

2. *Show that the triple above can be proved using the rules of Hoare logic.*

Let us assume that we replace the standard Hoare rule for assignment by the rule

$$\frac{}{\{P\}x := e\{\exists v,\ e[x \leftarrow v] = x \wedge P[x \leftarrow v]\}}$$

3. *Show that the triple* $\{P[x \leftarrow e]\}x := e\{P\}$ *can be proved with the new set of rules.*