Floyd-Hoare Logic for Program Verification

Julio Mariño

Rigorous Software Development
EUROPEAN MASTER IN SOFTWARE ENGINEERING/ MASTER IN SOFTWARE AND SYSTEMS
Universidad Politécnica de Madrid/IMDEA Software

October 2015

motivation

why Floyd-Hoare is so important

- Classical program verification consists in, given a piece of software written in one particular language and one property written in a certain kind of logic, *proving* that the execution of the program satisfies the property.
- Compared to correctness by construction (as in Event-B), classical program verification is the hard way to rigorous SW development because:
  - we have no control on *how* the code has been written (style, complexity, coding standards...), and
  - formalizing the semantics of *all* the constructs in real programming languages can be difficult.
- However, the basic techniques for program verification underlie many techniques and tools for program analysis so it is essential to gain certain familiarity to them.
- Hoare logic is one of the (many) valuable contributions of C.A.R. Hoare to computer science.
- The term "Floyd-Hoare" is due to the pioneering – but somehow informal – work by Robert W. Floyd: "Assigning Meanings to Programs", 1967.
- The practical work on this topic will be carried out using Dafny, a language and semiautomatic program verifier which allows to develop *certified* a.k.a. *proof-carrying* code making use of the programmer's annotations and an automated theorem prover.

POLITÉCNICA

motivation

some Java examples

```
public static final int square (int n) {
  int r = 0;
  int s = 1;
  int t = n;
  while (t > 0) {
    r = r + s;
    s = s + 2;
    t = t - 1;
  }
  return r;
}
```

motivation

some Java examples (ii)

```
public static final int euclid (int a, int b) {
  // a, b > 0
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
}
```

the need for a programming logic

- we have already proved that these Java programs have a "mathematical meaning" by proving things about mathematical formulae with a similar structure.
- but this is not really what we want:
  - data in a programming language – i.e. numbers, arrays, etc. – are not exactly the same as their mathematical relatives – natural numbers, sets, etc.
  - (imperative) programming languages have structures – assignment, sequential composition, loops, etc. – which do not have a clear equivalence in mathematics.
- we need a tool that combines mathematics and code in a single formal system, so that we can actually prove things about programs
- this is the purpose of Floyd-Hoare logic, and of every programming logic
- these ideas underlie other formal systems (e.g. Event-B)
- at the end of this unit we will be able to reason about, and prove properties of Java programs as the previous ones and those making use of static arrays – e.g. sorting routines.
- dealing with dynamic memory is harder and requires a more elaborate logic.

a simple programming language

while programs

$$
\begin{array}{lcll}
E & ::= & x \mid n \mid E_1 + E_2 \mid E_1 - E_2 \mid \ldots & \text{(ARITHMETIC EXPRESSIONS)} \\
B & ::= & E_1 = E_2 \mid E_1 \geq E_2 \mid \ldots & \text{(BOOLEAN EXPRESSIONS)} \\
P & ::= & \texttt{skip} \mid x := E \mid P_1 ; P_2 \mid & \\
 & & \texttt{if}\, B\, \texttt{then}\, P_1\, \texttt{else}\, P_2 \mid \texttt{while}\, (B)\, P & \text{(PROGRAMS)}
\end{array}
$$

- this is a simple "model language" but expressive enough to represent the two Java programs shown before.
- `skip` represents the empty program that does nothing.
- we are not considering arrays yet.

Hoare triples

- A Hoare triple $\{s_1\}\,P\,\{s_2\}$ says that if program P terminates when started in state s_1, then the resulting state will be s_2.
- A program state is a partial mapping of (arithmetic) variables to their values:

  $$State = Var \nrightarrow \mathbb{Z}$$

  Example: $\{x \mapsto 7, y \mapsto 42\}\, x := x + y\, \{x \mapsto 49, y \mapsto 42\}$
- A generalized state is a logic formula whose free variables are program variables. As program states can be trivially encoded as generalized ones, by using equality, we will routinely use generalized states in Hoare triples, which will result in better expressiveness and a simpler presentation of the deduction rules:

  Example: $\begin{array}{lll} \{x = 7 \wedge y = 42\} & x := x + y & \{x = 49 \wedge y = 42\} \\ \{x > 0 \wedge y > 0\} & x := x + y & \{x > 0\} \end{array}$

  A Hoare triple $\{\varphi_1\}\,P\,\{\varphi_2\}$ says that if program P terminates when started in a state *satisfying* φ_1, then the resulting state will satisfy φ_2.
- With a small notational abuse, we will consider BOOLEAN EXPRESSIONS valid generalized states.
- Observe that the concept of Hoare triple is *generic*, it can be easily adapted to different logics and programming languages.

- The purpose of the proof system is to prove *valid* Hoare triples.
- The proof system is *natural* and *uniform*: for each syntax rule to derive a program there is a single deductive rule.
- The axioms are given by the following rule scheme:

$$\frac{}{\{\varphi\}\,\texttt{skip}\,\{\varphi\}}\ \text{AX}$$

that is, for every formula φ, executing `skip` in a state satisfying φ will lead to an ending state satisfying φ.

- The simplest way of composing two proofs/programs is by sequential composition:

$$\frac{\{\varphi_1\}\,P_1\,\{\psi\} \qquad \{\psi\}\,P_2\,\{\varphi_2\}}{\{\varphi_1\}\,P_1;\ P_2\,\{\varphi_2\}}\ \text{SEQ}$$

That is, if P_1 ends in a state satisfying ψ when started in (a state satisfying) φ_1, and P_2 ends in φ_2 when started in ψ, then $P_1;\ P_2$ is a program that ends in φ_2 when started in φ_1.
Observe the similarities to the definition of the composition of two binary relations.

Hoare logic

carrying on

- Conditionals are also quite easily justified:

$$\frac{\{\varphi_1 \wedge B\}\, P_1\, \{\varphi_2\} \qquad \{\varphi_1 \wedge \neg B\}\, P_2\, \{\varphi_2\}}{\{\varphi_1\}\, \texttt{if}\,(B)\,\texttt{then}\, P_1\, \texttt{else}\, P_2\, \{\varphi_2\}}\ \textsc{seq}$$

  That is, if P_1 ends in a state satisfying φ_2 when started in φ_1 and B holds, and P_2 ends in φ_2 when started in φ_1 and B does not hold, then $\texttt{if}\,(B)\,\texttt{then}\, P_1\, \texttt{else}\, P_2$ is a program that ends in φ_2 when started in φ_1.

- Observe that the COND rule takes B both as a program expression and as a logical formula. We allow this notational abuse by economy of language.
- Structural rules: Rules SEQ and COND require that some of the states in the subproofs are *exactly* the same. As in sequent-based systems, some rules are provided that help in adapting the states in subproofs to the shapes required by a larger proof:

$$\frac{\{\varphi_1\}\, P\, \{\psi\} \qquad \psi \Rightarrow \varphi_2}{\{\varphi_1\}\, P\, \{\varphi_2\}}\ \textsc{post. weak.}$$

$$\frac{\{\psi\}\, P\, \{\varphi_2\} \qquad \varphi_1 \Rightarrow \psi}{\{\varphi_1\}\, P\, \{\varphi_2\}}\ \textsc{prec. strength.}$$

  The rule above is called POSTCONDITION WEAKENING and the one below PRECONDITION STRENGTHENING. Both of them should be obvious.

POLITÉCNICA

Hoare logic

structural rules in use

- Exercise: Prove the following rules:

$$\frac{\{\psi\}\, P\, \{\phi\} \qquad \varphi_1 \Rightarrow \psi \wedge \phi \Rightarrow \varphi_2}{\{\varphi_1\}\, P\, \{\varphi_2\}}\ \text{STRUCT}$$

$$\frac{\{\varphi_1 \wedge B\}\, P_1\, \{\psi\} \qquad \{\varphi_1 \wedge \neg B\}\, P_2\, \{\phi\}}{\{\varphi_1\}\texttt{if}\,(B)\,\texttt{then}\,P_1\,\texttt{else}\,P_2\{\psi \vee \phi\}}\ \text{COND'}$$

- What happens when we *change* the program state?

$$\begin{array}{rcl} \{x = 42\} & x := 7 & \{x = 7\} \\ \{x = 42 \wedge y = 7\} & x := y & \{x = 7 \wedge y = 7\} \\ \{x = 42 \wedge y = 7\} & x := x + y & \{x = 49 \wedge y = 7\} \end{array}$$

- Hoare found a common pattern (!):

$$\{\varphi[e/x]\}x := e\{\varphi\}$$

- The rule seems counterintuitive, but it works:

$$\begin{array}{rcl} \{x = 7[7/x]\} & = & \{true\}(x = 42 \Rightarrow true) \\ \{(x = 7 \wedge y = 7)[7/x]\} & = & \{true \wedge y = 7\} \\ \{(x = 49 \wedge y = 7)[x + y/x]\} & = & \{y = 7 \wedge x + y = 49\} \\ & & (y = 7 \wedge x + y = 49) \Rightarrow x = 42 \end{array}$$

- Note that the precondition can be obtained from the postcondition, but equation solving may be necessary to take parctical advantage of it.

Hoare logic

loops

- $$\frac{\{\varphi \wedge B\}\, P\, \{\varphi\}}{\{\varphi\}\, \texttt{while}\, (B)\, P\, \{\varphi \wedge \neg B\}}\ \textsc{while}$$

- φ is an invariant: if it is assumed to hold *immediately before* executing the loop, then we know it holds *right after*.
- We also know that, if the loop terminates, the *loop condition* no longer holds.
- The problem is, we have no recipe for inferring the *right* invariant.
- This is because, in general, we cannot know how many times the loop is going to execute – if we knew it, we could just *unroll* it and make it a *big* conditional of conditionals, etc.
- However, it is possible to *approximate* invariants, and this is actually an active research topic.
- To summarize, the WHILE rule is the only one that requires some kind of *creativity* to be applied. This is the crucial piece of information one has to provide to certain automated program verification tools (e.g. Dafny).
- Also, remember that, for this to be practical, termination of the loop needs to be established.

a practical case

squares as sums of odd numbers

```
1  public static final int square (int n) {
2    int r = 0;
3    int s = 1;
4    int t = n;
5    while (t > 0) {
6      r = r + s;
7      s = s + 2;
8      t = t - 1;
9    }
10   return r;
11 }
```

- Writing a whole proof in mathematical style for even a program this small can take a lot of paper....
- Fortunately, the uniformity of the proof system and the associativity of the SEQ rule makes it possible a more compact representation: just annotate the program with formulae inbetween the sentences and the proof can be automatically inferred from the code structure...

a practical case (II)

squares as sums of odd numbers

```
public static final int square (int n) {
  // n >= 0 (assumed)
  int r = 0;
  int s = 1;
  int t = n;
  while (t > 0) {
    r = r + s;
    s = s + 2;
    t = t - 1;
  }
  // r = n * n (desired)
  return r;
}
```

- We use comments as states: a (sub)program enclosed in comments can be seen as a Hoare triple.
- We start by stating the program's "contract".

POLITÉCNICA

a practical case (III)

squares as sums of odd numbers

```
public static final int square (int n) {
  // n >= 0 (assumed)
  int r = 0;
  int s = 1;
  int t = n;
  // r == 0 && s == 1 && t == n
  // r == (n - t) * (n - t) (derived)
  while (t > 0) {
    r = r + s;
    s = s + 2;
    t = t - 1;
  }
  // r == (n - t) * (n - t)
  // r == n * n (desired, derived from previous line)
  return r;
}
```

- We need some property that holds before and after the loop, so it can be used as an invariant for it.
- We can introduce more comments as far as we are consistent with the SEQ and structural rules.

a practical case (IV)

squares as sums of odd numbers

```
public static final int square (int n) {
  // n >= 0 (assumed)
  [...]
  // r == 0 && s == 1 && t == n
  // r == (n - t) * (n - t) (derived)
  // s == 2 * (n - t) + 1   (derived)
  // t >= 0                 (derived)
  while (t > 0)
  // t >= 0                 (invariant)
  // r == (n - t) * (n - t) (invariant)
  // s == 2 * (n - t) + 1   (invariant)
  {
    [ loop body ]
  }
  // t == 0 (t >= 0 & !(t > 0))
  // r == (n - t) * (n - t)
  // r == n * n             (desired, derived from previous lines)
  return r;
}
```

- something must be said about `s`, as it is used to update the value for `r`...
- we need to note that `t` cannot become negative

a practical case (V)

squares as sums of odd numbers

- Finally, we prove that the loop body preserves the invariant:

```
// t >  0                    (we are inside the loop!)
// r == (n - t) * (n - t) (invariant)
// s == 2 * (n - t) + 1   (invariant)
// r + s == (n - t) * (n - t) + 2 * (n - t) + 1 <=>
// r + 2 * (n - t) + 1 == (n - t) * (n - t) + 2 * (n - t) + 1 <=>
// r == (n - t) * (n - t) (as desired)
r = r + s;
// t > 0
// r == (n - t) * (n - t) + 2 * (n - t) + 1
// s + 2 == 2 * (n - t) + 3 <=> s == 2 * (n - t) + 1
s = s + 2;
// t - 1 >= 0 <=> t >= 1 <=> t > 0
// r == (n-t + 1) * (n-t + 1) == (n-t) * (n-t) + 2 * (n-t) + 1
// s == 2 * (n - t + 1) + 1 == 2 * (n - t) + 3
t = t - 1;
// t >= 0                    (invariant)
// r == (n - t) * (n - t) (invariant)
// s == 2 * (n - t) + 1   (invariant)
```