Using logic to specify and verify programs

Jonathan P. Bowen

www.cs.ucl.ac.uk/staff/J.Bowen/GS03
Overview

• Background
• Core programming language
• Hoare triples $\langle \phi \rangle P \langle \psi \rangle$
• Partial correctness $\models_{\text{par}}$
• Total correctness $\models_{\text{tot}}$
• Proof rules for partial correctness
Verification method (1)

- **Proof-based** – construct a proof rather than exhaustively check state (cf. model checking).

- **Semi-automatic** – many steps mechanical but some need intelligence (e.g., up to 95% automatic in practice).

- **Proof-oriented** – verify program properties rather than full specification of the behaviour.
Verification method (2)

- **Application domain** – sequential transformational programs (no concurrency, takes input and produces output).
- **Pre/post-development** – use during development for small critical sections of code.
Motivation

- **Documentation** – formal specification is an important part of the documentation (if available!)
- **Time-to-market** – testing is expensive and time-consuming; a formal spec helps reduce testing time/cost
- **Refactoring** – a formal spec can help with reuse (since we know what the program does)
- **Certification audits** – a verification proof might be part of the warranty for critical software
Software verification framework

• Convert informal requirements $R$ into an “equivalent” logical formula $\phi_R$.
• Write a program $P$ designed to implement $\phi_R$ in the available programming environment.
• Prove that program $P$ satisfies the formula $\phi_R$ in some logical framework.
Problems

- Some constraints may be design decisions (e.g., interfaces, data types)
- The specification may evolve over time
- The specification may be incomplete (especially for larger projects)
- Formalizing $R$ may lead to revisions due to ambiguities & undesirable consequences
- The process of relating $R$ and $\phi_R$ is necessarily informal
<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Involves</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Formal Specification</td>
<td>Formal notation used for specifying requirements only; no analysis/proof</td>
</tr>
<tr>
<td>1</td>
<td>Formal Development / Verification</td>
<td>Proving properties and applying refinement calculus</td>
</tr>
<tr>
<td>2</td>
<td>Machine Checked Proofs / Model checking</td>
<td>Use of theorem prover/checker tool to prove consistency/integrity.</td>
</tr>
</tbody>
</table>
Cost of proofs

- Mathematics – simple theorems, deep proofs (decades or centuries)
- Cf. software – complicated specs & programs, shallow proofs (B, 90–95% automated, 5–10% manual, weeks or months).

Fermat’s Last Theorem (in Toulouse)
\[ a^n + b^n \neq c^n \quad (n>2) \]

— Pierre de Fermat (1601–1635)
Hand vs. machine checked proofs

Blackboard at workshop!
Overview

• Background

• **Core programming language**

• Hoare triples \( \langle \phi \rangle P \langle \psi \rangle \)

• Partial correctness \( \vdash_{\text{par}} \)

• Total correctness \( \vdash_{\text{tot}} \)

• Proof rules for partial correctness
Core programming language (1)

Integer expressions:

- \( E ::= n \mid x \mid (–E) \mid (E + E) \mid (E – E) \mid (E * E) \)

- \( n \) is an integer \( \{…, –2, –1,0,1,2,… \} \) (set \( A \))

- \( x \) is any variable (\( \text{var} \))

- – negation binds most tightly, then
  * multiplication, then – subtraction and + addition
Core programming language (2)

Boolean expressions:

- $B ::= \text{true} \mid \text{false} \mid (! B) \mid (B \& B) \mid (B \| B) \mid (E < E)$
- $!$ is negation, $\&$ is conjunction, $\|$ is disjunction
- Can be expanded with macro style operators ("syntactic sugar"). E.g.:
  
  $E_1 == E_2 \equiv !(E_1 < E_2) \& !(E_2 < E_1)$
  
  $E_1 != E_2 \equiv !(E_1 == E_2)$
Core programming language (3)

Commands:

• \[ C ::= x = E \mid C ; C \mid \text{if } B \{ C \} \ \text{else } \{ C \} \mid \text{while } B \{ C \} \]

• \{…\} marks blocks of code

• \( x = E \) - assignment (atomic command)

• \( C_1 ; C_2 \) - sequential composition
  First \( C_1 \) is executed, then if it terminates \( C_2 \)

• \( \text{if } B \{ C_1 \} \ \text{else } \{ C_2 \} \) - conditional
  If \( B \) evaluates to true, \( C_1 \) is executed, otherwise \( C_2 \)

• \( \text{while } B \{ C \} \) - iteration
  If \( B \) evaluates to false, the command terminates, otherwise \( C \) is executed and the command is repeated
Example

Factorial $x!$ of a natural number (defined by induction):

$$0! \equiv 1$$

$$(x + 1)! \equiv (x + 1) \cdot x!$$

Program **Fac1** (on termination, $y = x!$):

```plaintext
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z
}
```
Overview

• Background
• Core programming language

• Hoare triples $\langle \phi \rangle \ P \ \langle \psi \rangle$

• Partial correctness $\vdash_{\text{par}}$
• Total correctness $\vdash_{\text{tot}}$
• Proof rules for partial correctness
Hoare triples

\((\phi) \ P \ (\psi)\)

If the program \(P\) is run in a state that satisfies \(\phi\), then the state resulting from \(P\)’s execution will satisfy \(\psi\).

\(\phi\) – precondition (assumption)

\(\psi\) – postcondition (guarantee)

\(\phi\) and \(\psi\) are predicates or “assertions”.

They specify the program behaviour.

Concept named after Tony Hoare (aka Hoare logic).
Example

$$\{ x > 0 \} \ P \ { y \cdot y < x \}$$

Calculates a number $y$ whose square is less than $x$.

Note that there is no guarantee if $x \leq 0$ (the program can do anything, it may not even terminate).
Definition – Hoare triple \( (\phi) \ P \ (\psi) \)

\( \phi \) is the precondition of \( P \).
\( \psi \) is the postcondition of \( P \).

The state is a function \( l \) from variables to integers \((l : \text{var} \rightarrow A)\).

A state \( l \) satisfies \( \phi \) (\( l \) is a \( \phi \)-state or \( l \models \phi \)) iff \( M \models_l \phi \) (model \( M \) satisfies formula \( \phi \) in environment \( l \) – see slide 22 of week1 lecture 3) where \( M \) has as set \( A \) all integers.

Note: variables bound by quantifiers in \( \phi \) and \( \psi \) must not occur in \( P \).
State examples

For state \( l \) including \( \{x \mapsto -2, \ y \mapsto 5, \ z \mapsto -1\} \),
do the following hold?

\[
l \models \neg(x + y < z)
\]
\[
l \models y - x \cdot z < z
\]
\[
l \models \forall u (y < u \Rightarrow y \cdot z < u \cdot z)
\]

Holds since \( x + y = 3 \) and \( z = -1 \)

Does not hold since LHS = 3

Does not hold; e.g., \( u = 6 \)
Program examples

Do the following hold?

\[
\{ x > 0 \} \ y = 0 \{ y \cdot y < x \}
\]

\[
\{ x > 0 \}
\]

\[
y = 0;
\]

\[
\text{while} \ (y \cdot y < x) \ \{ \ y = y+1 \ \};
\]

\[
y = y-1
\]

\[
\{ y \cdot y < x \}
\]
Program examples (2)

\[ (x > 0) \; y = 0 \; (y \cdot y < x) \]

\( y \cdot y \) is 0 in the postcondition since \( y = 0 \).
Since \( x > 0 \) in the precondition,
\( y \cdot y < x \) holds after the program.
This is not a very useful program.
Note that the specification is non-deterministic.
Different programs can satisfy the same spec.
Program examples (3)

\( \{ x > 0 \} \)

\[ y = 0; \text{while } (y \cdot y < x) \{ y = y+1 \}; y = y-1 \]

\( \{ y \cdot y < x \} \)

This finds the greatest \( y \) whose square is less than \( x \).

Can you think of a stronger postcondition to specify this more exactly?
Proof of correctness?

• We can reason about such programs informally
• In fact, programmers (should) do this all the time
• But what if we want a formal proof?
• We can do proofs in propositional/predicate logic 😊
• E.g., to prove $\phi \Rightarrow \psi$, assume $\phi$ and show $\psi$ holds
• Now we have triples, logical formulas $\phi$ and $\psi$, and a piece of code $P$
• We want “compositional” proofs over $P$’s structure
• Very important for larger proofs (e.g., with subroutines)
Overview

• Background
• Core programming language
• Hoare triples $\langle \phi \rangle P \langle \psi \rangle$

• **Partial correctness** $\models_{\text{par}}$
• **Total correctness** $\models_{\text{tot}}$
• Proof rules for partial correctness
Partial correctness (1)

• For $\langle \phi \rangle P \langle \psi \rangle$ what should happen when $P$ does not terminate (e.g., an infinite loop)?

• Under “partial correctness”, $\langle \phi \rangle P \langle \psi \rangle$ is satisfied if, for all states that satisfy $\phi$, the state resulting from $P$’s execution satisfies $\psi$, provided $P$ terminates.

• I.e., $\models_{\text{par}} \langle \phi \rangle P \langle \psi \rangle$ holds, where $\models_{\text{par}}$ is the satisfaction relation for partial correctness.
Partial correctness (2)

- Partial correctness is a rather weak requirement.
- Any program that does not terminate satisfies its specification.
- E.g., `while true { skip }` (where `skip ≡ x=x` for any variable `x`) never terminates and this satisfies all partial correctness specs.
- Solution: “total correctness”.

Overview

• Background
• Core programming language
• Hoare triples $\langle \phi \rangle P \langle \psi \rangle$
• Partial correctness $\models_{\text{par}}$
  • **Total correctness** $\models_{\text{tot}}$
• Proof rules for partial correctness
Total correctness (1)

Under “total correctness”, $\langle \phi \rangle P \langle \psi \rangle$ is satisfied if, for all states that satisfy $\phi$, the state resulting from $P$’s execution satisfies $\psi$, and $P$ is guaranteed to terminate.

I.e., $\models_{\text{tot}} \langle \phi \rangle P \langle \psi \rangle$ holds, where $\models_{\text{tot}}$ is the satisfaction relation for total correctness.

A program that loops forever for all inputs cannot satisfy any valid spec under total correctness.
Total correctness (2)

• Total correctness is of more practical use than partial correctness.
• So why have partial correctness?
• Often it is easier to prove partial correctness first and then prove termination.

• **Hint:** This is a good strategy for proving program correctness. 😊
Example (1)

Consider a program \textbf{Succ}:
\begin{verbatim}
a = x+1;
if (a–1 == 0) {
    y = 1
} else {
    y = a
}
\end{verbatim}

Consider a spec \( \top \) \textbf{Succ} \( \left( y = x+1 \right) \)

(Note that \( \top \) indicates the true predicate)
Example (2)

\[ a = x+1; \text{if } (a-1 == 0) \{y = 1\} \text{ else } \{y = a\}\]

\[\top\quad \textbf{Succ} \quad (y = x+1)\]

Is this satisfied under:

1. Partial correctness?
2. Total correctness?

Note there are no loops in the program.

With \(x\) as the input and \(y\) as the output, this is a rather roundabout successor function.
Example (3) – Fac1 program

\[ y = 1; \quad z = 0; \]
\[ \text{while} \ (z != x) \ \{ \ z = z + 1; \quad y = y \times z \ \} \]

When does this terminate?

Which of the following hold?

\( \models_{\text{par}} \left( x \geq 0 \right) \text{Fac1} \left( y = x! \right) \)

\( \models_{\text{par}} \left( \top \right) \text{Fac1} \left( y = x! \right) \)

\( \models_{\text{tot}} \left( x \geq 0 \right) \text{Fac1} \left( y = x! \right) \)

\( \models_{\text{tot}} \left( \top \right) \text{Fac1} \left( y = x! \right) \)
Soundness and completeness (1)

If the partial correctness of $\langle \phi \rangle P \langle \psi \rangle$ can be proved using partial-correctness calculus, the sequent $\vdash_{\text{par}} \langle \phi \rangle P \langle \psi \rangle$ is valid.

Similarly for total correctness.
Soundness and completeness (2)

\[ \vdash_{\text{par}} (\phi) \quad P (\psi) \quad \text{holds whenever} \quad \not\vdash_{\text{par}} (\phi) \quad P (\psi) \quad \text{is valid.} \]

\[ \vdash_{\text{con}} (\phi) \quad P (\psi) \quad \text{is valid whenever} \quad \vdash_{\text{con}} (\phi) \quad P (\psi) \quad \text{holds.} \]

\[ \vdash_{\text{tot}} (\phi) \quad P (\psi) \quad \text{holds whenever} \quad \not\vdash_{\text{tot}} (\phi) \quad P (\psi) \quad \text{is valid.} \]

\[ \vdash_{\text{tot}} (\phi) \quad P (\psi) \quad \text{is valid whenever} \quad \vdash_{\text{tot}} (\phi) \quad P (\psi) \quad \text{holds.} \]
Example – Fac2

Consider an alternative program:

\[ y = 1; \text{while } (x \neq 0) \{ y = y \times x; x = x - 1 \} \]

Note that \( x \) is “consumed” (destroyed).

We need to remember the initial value – use logical variable \( x_0 \) to record this value.

\[ \{ x = x_0 \land x \geq 0 \} \quad \text{Fac2} \quad \{ y = x_0! \} \]

\( x_0 \) does not occur in \text{Fac2} & cannot be modified by it.
Example – Sum

Consider:

\[ z = 0; \text{while} \ (x > 0) \ {\{z = z + x; \ x = x - 1\}} \]

\[ 0+1+2+\ldots+n = \frac{n \cdot (n+1)}{2} \text{ (by induction)} \]

What is a spec for this program?
Logical variables

• Variables like $x_0$ are *logical variables* because they only occur in the logical formulas (precondition and postcondition).

• For a Hoare triple $\langle \phi \rangle \ P \ \langle \psi \rangle$ the set of logical variables are those that are free in $\phi$ or $\psi$, and not occurring in $P$. 
Overview

• Background
• Core programming language
• Hoare triples \( [\phi] P [\psi] \)
• Partial correctness \( \vdash_{\text{par}} \)
• Total correctness \( \vdash_{\text{tot}} \)

• Proof rules for partial correctness
Proof calculus – partial correctness

- Proof calculus of R. Floyd and C. A. R. Hoare
- See summary on page 270 of Huth & Ryan book
- Proof rules:

\[ (φ) \quad C_1 \quad (η) \quad (η) \quad C_2 \quad (ψ) \]

Composition

\[ (φ) \quad C_1 ; C_2 \quad (ψ) \]

The postcondition of the code fragment \( C_1 \) is the same as the precondition of \( C_2 \).

\( η \) is an intermediate midcondition.
Proof rule: Assignment (1)

• No premises – \( \therefore \) an axiom of the logic.

• \( \psi [E/x] \) has all free occurrences of \( x \) replaced with \( E \).

\[
\begin{align*}
\text{Assignment} \\
\langle \psi [E/x] \rangle & \quad x = E \quad \langle \psi \rangle
\end{align*}
\]

This rule is best applied backwards from the postcondition.
Proof rule: Assignment (2)

- The rule can be applied completely mechanically (unlike while for example).
- Unproblematic provided pre/postconditions quantify over logical variables only — recommended!
- Why not use $\phi(x = E \phi[E/x])$?
- Consider if $\phi$ is $x = 6$ and $E$ is 5, for example.
- The assignment “$x = 5$” should give a postcondition of $x = 5$, but $\phi[E/x]$ is the formula $5 = 6$, which is the same as $\bot$ (i.e., false).
Proof rule: Assignment (3)

Suppose $P$ is “$x = 2$”. Do the following hold?

- $\begin{align*} 2 = 2 \end{align*} \quad \begin{align*} 2 = 2 \end{align*} \quad P \quad \begin{align*} x = 2 \end{align*}$
- $\begin{align*} 2 = 4 \end{align*} \quad \begin{align*} 2 = 4 \end{align*} \quad P \quad \begin{align*} x = 4 \end{align*}$
- $\begin{align*} 2 = y \end{align*} \quad \begin{align*} 2 = y \end{align*} \quad P \quad \begin{align*} x = y \end{align*}$
- $\begin{align*} 2 > 0 \end{align*} \quad \begin{align*} 2 > 0 \end{align*} \quad P \quad \begin{align*} x > 0 \end{align*}$

Note: These are all instances of the axiom.

In general, $\begin{align*} \bot \end{align*} \quad \begin{align*} x = E \end{align*} \quad \begin{align*} \psi \end{align*}$ — why?

$\begin{align*} \psi [E/x] \end{align*} \quad \begin{align*} x = E \end{align*} \quad \begin{align*} \psi \end{align*}$
Proof rule: Assignment (4)

Suppose $P$ is "$x = x+1$". Do the following hold?

• $\langle x+1 = 2 \rangle \ P \ \langle x = 2 \rangle$
• $\langle x+1 = y \rangle \ P \ \langle x = y \rangle$
• $\langle x+1+5 = y \rangle \ P \ \langle x+5 = y \rangle$
• $\langle x+1 > 0 \land y > 0 \rangle \ P \ \langle x > 0 \land y > 0 \rangle$

Note: These are also all instances of the axiom. Preconditions obtained from this rule can often be simplified further for readability.

$\langle \psi [E/x] \rangle \ x = E \ (\psi)$
Proof rule: If-statements

- Split into two sub-goals for the $B$ and $\neg B$ cases.
- Knowledge of $B$ and $\neg B$ is typically crucial in the sub-proofs since $\phi$ is normally unrelated.

\[
\begin{align*}
(\phi \land B) & \quad C_1 & \quad (\psi) & \quad (\phi \land \neg B) & \quad C_2 & \quad (\psi)
\end{align*}
\]

If-statement

\[
(\phi) \text{ if } B \{ C_1 \} \text{ else } \{ C_2 \} (\psi)
\]

- The postcondition of the code fragment $C_1$ is the same as that for $C_2$. 
Proof rule: While-statements

• Arguably the most complicated rule (despite initial appearances) because of looping.
• Number of times of looping is difficult to predict.

\[
\begin{align*}
&\langle \psi \land B \rangle \quad C \quad \langle \psi \rangle \\
\hline
&\langle \psi \rangle \quad \text{while} \quad B \quad \{C\} \quad \langle \psi \land \neg B \rangle
\end{align*}
\]

Partial-while

• The critical component is the “invariant” \( \psi \), which must be true at the start and end of execution of \( C \), and must normally be determined manually (using intelligence!).
Proof rule: Implied

• Precondition strengthened (assume more).
• Postcondition weakened (conclude less).

• Sequent $\vdash_{\text{AR}} \phi' \Rightarrow \phi$ is valid iff there is a natural deduction predicate calculus (+ arithmetic) proof

\[\vdash_{\text{AR}} \phi' \Rightarrow (\phi) \quad C \quad (\psi) \quad \vdash_{\text{AR}} (\psi) \Rightarrow \psi' \quad \text{Implied} \]

\[ (\phi') \quad C \quad (\psi') \]

• Acts as a link between predicate logic and program logic.
Hoare logic summary

• Core programming language
• Hoare triples: $\langle \phi \rangle P \langle \psi \rangle$
• Partial/total correctness: $\vdash_{\text{par}}$ and $\vdash_{\text{tot}}$
• Simple set of partial correctness proof rules
• However, not easy to use in program examples in this form
• Next: consider program proofs (verification)
Conclusion

• This week:
  – Using logic to specify programs
  – Using logic to verify programs

• Next week:
  – Static analysis
  – JML (Java Modeling Language)
  – ESC/Java2 tool
  – Conclusion (state of the art)
Reading and exercises

• Read Chapter 4 of Huth and Ryan up to section 4.3.1.

• Do questions 1-4 of the book’s online tutor for Chapter 4 under

• Do selected (starred) exercises:
  – 4.1: 1; 4.2: 1, 2; 4.3: 1(a,c), 2