Program verification

Assertional semantics of a program

“Meaning” of a program:
- relation between its inputs and outputs;
- specified by input assertions (pre-conditions)
  and output assertions (post-conditions)
- assertions are predicate-logic formulae;
  atomic propositions are program variables

Example assertion:
\[ x > 20 \land y \leq 15 \]
where \( x \) and \( y \) are variables

Assertional semantics (cont)

Key ideas:
- Formalize the meaning of each type of statement using a deductive system of axioms and inference rules.
- Program can be proved correct by constructing a proof in this system.
- Program can even be synthesized from its axiomatic specification by searching for such a proof, using the structure of the proof to construct the program.

Contributions to programming:
- Proofs of program correctness
- Notion of invariants to aid in documentation and understanding

Hoare triples

Assertions of the form:
\( \{ P \} \ S \ { Q \} \)
where \( P \) & \( Q \) are assertions and \( S \) is a statement.

Interpretation: If \( P \) is true before execution of \( S \), and if \( S \) terminates, then \( Q \) is true thereafter.

Example:
\( \{ x = 5 \} \ x := x + 1 \ { x = 6 \} \)

Semantics of assignment

Given an assignment of the form:
\[ x := E \]
and a post-condition \( P \).

Axiomatic definition of assignment is:
\[ \{ P [x \rightarrow E] \} \ x := E \ \{ P \} \]
where \( P [x \rightarrow E] \) is the consistent substitution of \( E \) for all occurrences of \( x \) in \( P \)

Example:

Theorem: \( \{ k = 5 \} \ k := k + 1 \ { k = 6 \} \)

Proof:
- Let \( P = \{ k = 6 \} \) be the post-condition
- By the assignment axiom, the pre-condition must be: \( P [k \rightarrow k + 1] \)
  \[ \{ k = 6 \} (k + 1 = 6) = \{ k = 5 \} \]
Example:

**Theorem:** \{ a > 0 \} a := a – 1 \{ a \geq 0 \}

**Proof:**
- Let \( P = \{ a \geq 0 \} \) be the post-condition
- Then the pre-condition must be \( P \{ a \rightarrow a – 1 \} \)
- \( P \{ a \rightarrow a – 1 \} = (a – 1) \geq 0 \)
- Thus:
  \( (a – 1) \geq 0 \) \( a := a – 1 \{ a \geq 0 \} \)
- Note: this is not exactly what we wanted

Inference rules

Sometimes a pre-condition is *stronger* than what is proved by applying the assignment axiom.
- This is OK.
- The following inference rule justifies replacing the “proved” pre-condition with the stronger one:

\[
\begin{align*}
  P & \Rightarrow Q, \quad \{ Q \} S \{ R \} \\
  \hline
  \{ P \} S \{ R \} \quad \text{[strengthenPre]}
\end{align*}
\]

Example (continued)

Strengthen pre-condition to “finish” proving \( \{ a > 0 \} a := a – 1 \{ a \geq 0 \} \)

from:
\( \{ (a – 1) \geq 0 \} a := a – 1 \{ a \geq 0 \} \)

**Proof (continued):**
- Note: \( a > 0 \Rightarrow a \geq 1 \Rightarrow (a – 1) \geq 0 \)
- Thus: \( a > 0 \Rightarrow a – 1 \geq 0 \)
- Then apply pre-condition strengthening rule.

Weakening post-conditions

It may also be useful to “weaken” a post-condition.

The following rule justifies such an action:

\[
\begin{align*}
  \{ P \} S \{ Q \}, \quad Q \Rightarrow R \\
  \hline
  \{ P \} S \{ R \} \quad \text{[weakenPost]}
\end{align*}
\]

Semantics of read and write

I/O commands:
- assume existence of input and output files
- We use “IN = …” and “OUT = …” to indicate contents of these files in assertions

Axiom for read:
\( \{ \text{IN} = [\kappa] L \land P \{ V \rightarrow \kappa \} \} \text{read} V \{ \text{IN} = L \land P \} \)

Axiom for write:
\( \{ \text{OUT} = L \land E = \kappa \land P \} \text{write} E \{ \text{OUT} = L[\kappa] \land E = \kappa \land P \} \)

Semantics of sequencing

Sequential composition of statements requires the post-condition of the first to establish the pre-condition of the second.

\[
\begin{align*}
  \{ P \} S_1 \{ Q \}, \quad \{ Q \} S_2 \{ R \} \\
  \hline
  \{ P \} S_1 ; S_2 \{ R \} \quad \text{[sequence]}
\end{align*}
\]
Semantics of conditionals

The “if” command involves a choice between options.

\[
\begin{align*}
\{ P \land B \} & \quad S_1 \quad \{ Q \} \\
\{ P \land \neg B \} & \quad S_2 \quad \{ Q \} \\
\{ P \} & \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end if } \{ Q \}
\end{align*}
\]

Semantics of conditionals (cont)

When there is no else part, the rule simplifies to:

\[
\begin{align*}
\{ P \land B \} & \quad S \quad \{ Q \} \\
\{ P \land \neg B \} & \Rightarrow Q \\
\{ P \} & \text{if } B \text{ then } S \text{ end if } \{ Q \}
\end{align*}
\]

Semantics of loops

Proof rule organized around loop invariant
– Assertion which:
  • Must hold initially,
  • Must hold after each execution of loop body, and
  • Combined with exit condition, must imply loop post-condition
– Captures the “essence” of computation in the loop
– Requires insight to develop

\[
\{ I \} \quad \text{while } B \text{ do } S \text{ end while } \{ I \land \neg B \}
\]

Example

\[
\begin{align*}
\{ N \geq 0 \} & \\
k & := N; \\
f & := 1; \\
\text{while } k > 0 \text{ do } \\
f & := f \times k; \\
f & := k - 1; \\
\text{end while } \\
\{ f = n! \}
\end{align*}
\]

Question: What is the invariant?

Example (continued)

<table>
<thead>
<tr>
<th>k</th>
<th>f</th>
<th>f*k!</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>120</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>120</td>
</tr>
</tbody>
</table>

Candidate invariant:
\[
I \equiv \{ N! = f \times k! \land k \geq 0 \}
\]

Example: Setting up invariant

Theorem:
\[
\begin{align*}
\{ N \geq 0 \} & \\
k & := N; \\
f & := 1; \\
\{ f\times k! = N! \land k \geq 0 \}
\end{align*}
\]

Proof:
\[
\begin{align*}
\{ N \geq 0 \} & \Rightarrow \\
\{ n! = n! \land N \geq 0 \} \\
k & := N; \\
\{ k! = N! \land k \geq 0 \} & \Rightarrow \\
\{ 1 \times k! = N! \land k \geq 0 \} \\
f & := 1; \\
\{ f\times k! = N! \land k \geq 0 \}
\end{align*}
\]

Observe: By tabulating values, we can discover a value that is invariant.
Example: Invariant preserved in body

**Theorem:**
\[
\{ I \land (k > 0) \} \Rightarrow \{ f * k! = N! \land k > 0 \} \Rightarrow \{ f * (k - 1)! = N! \land (k - 1) \geq 0 \} \Rightarrow \{ f = N! \} \land k = 0 \Rightarrow \{ f = N! \}
\]

**Guidelines for discovering invariant**

Loop invariant describes relationship among variables that does not change as loop executes:
- Variables may change values, but relationship remains the same
- Constructing table of values for variables often reveals a property that does not change
- Combining what has been computed with what has yet to be computed often yields a constant
- Expression related to test \((\mathcal{B})\) for the loop can usually be combined with assertion \((\neg \mathcal{B})\) to produce part of post-condition

Semantics of blocks

Procedure/constant declarations “pre-processed”:
- For each declared constant \(c\) with value \(N\), predicate \(\text{Const}\) has a conjunct: \(c = N\).
- For each declared procedure \(p\), predicate \(\text{Procs}\) has:
  - a conjunct: \(\text{body}(p) = B\), where \(B\) is the body of \(p\) and
  - a conjunct: \(\text{parameter}(p) = F\), where \(F\) is the formal parameter of \(p\) (if any).

Proof rule is:
\[
\text{Procs} \vdash (P \land \text{Const}) \land C \land (Q) \quad \text{[sequence]}
\]
\[
\{ P \} \text{ begin } C \text{ end } \{ Q \}
\]

Example: Assertion after loop

**Proof:**
\[
\{ I \land \neg (k > 0) \} \Rightarrow \{ f * k! = N! \land k \geq 0 \land k \leq 0 \} \Rightarrow \{ f * k! = N! \land k = 0 \} \Rightarrow \{ f = N! \}
\]

Algol-60 style blocks

Block declarations take the form:

- `Block \rightarrow \text{Declaration}^* \text{ begin } \text{Command} \text{ end}`
- \(\text{Declaration} \rightarrow \text{const Identifier = Expression} \mid \text{var Identifier : Type} \mid \text{procedure Identifier (Identifier : Type ) is Block}

Example:
```
declare
const x = 10;
var y : integer;
begin
read y; \ y := x + y;
write y
end
```

Non-recursive procedure calls

Calling non-recursive procedure with no parameter requires proving logical relation of assertions around the execution of procedure body.

More precisely:
\[
(P) \; B \; (Q), \; \text{body(proc)} = B \quad \{P\} \text{ proc } \{Q\}
\]
Example (procedure call)

```
declare
procedure square is
begin
  x := x * x
end
begin
  square
end
```

Here:
- **Procs** is the assertion `body(square) = (x := x * x)`
- **Const** is the assertion `true`
- Thus, we must show:
  \[
  \text{body(square)} = (x := x * x) \\
  \{ x \in \Pi \land \text{true} \} \quad \square \quad \{ x \in \Pi * \Pi \}
  \]
- To apply Call0, we must show:
  \[
  \{ x \in \Pi \land \text{true} \} \quad x := x * x \quad \{ x \in \Pi * \Pi \}
  \]

Example (procedure call)

```
declare
procedure increment( step : integer ) is
begin
  x := x + step
end
begin
  increment(y)
end
```

Procs \equiv \text{parameter(increment) = step} \land \text{body(increment) = (x := x + step)}

\text{Const} \equiv \text{true}

We must show:
\[
\text{parameter(increment) = step} \land \text{body(increment) = (x := x + step)} \\
\{ x = M, \ y = N, \ \text{true} \} \quad \text{increment(y)} \quad \{ x = M + N, \ y = N \}
\]

Proof

By rule for procedure invocation with parameters, we must show:
\[
\{ x = M \land \ \text{step} = N \} \quad x := x + \text{step} \quad \{ x = M + N \land \ \text{step} = N \}
\]

Recursive procedures

Prove a call to a recursive procedure is correct by assuming embedded calls to the procedure are correct.

More precisely:
\[
\{ P \} \ \text{proc} (Q) \quad \vdash \quad \{ P \} \ \text{C} (Q), \ \text{body(proc) = C} \quad \{ P \} \ \text{proc} (Q)
\]

Calls with parameters

Calling non-recursive procedure with parameter requires binding actual parameter to formal in pre- and post-condition.

More precisely:
\[
\{ P \} \ \text{B} (Q), \ \text{body(proc) = B}, \ \text{parameter(proc) = F} \\
\{ P \ \text{I} F \rightarrow E \} \quad \text{proc}(E) \quad \{ Q \ \text{I} F \rightarrow E \}
\]
Proving termination

Proof rules described thus far:
- show only partial correctness
  - i.e., \( (P) S (Q) \) means that when initiated with \( P \) true, if execution of \( S \) terminates, then \( Q \) will be true thereafter.
- proof of termination handled as separate problem

Termination not problem for some statements:
- e.g., assignments, i/o, and non-recursive procedure invocation

Must prove termination for while loops and invocation of recursively defined procedures

Showing termination of while loops

Two steps:
- Find a set \( W \) with a strict well-founded ordering
- Find a termination expression, \( E \), with the following properties:
  - Whenever control passes through the beginning of the iterative loop, the value of \( E \) is in \( W \)
  - \( E \) takes a smaller value with respect to > each time the top of the iterative loop is passed.

More precisely:
- \( P \land B \Rightarrow E \in W \)
- \( (P \land B \land E = a) \land C (a > E) \)

Example

```plaintext
read n;
k := 0; f := 1;
while k < n do
  k := k + 1; f := f * k
end while;
write f
```

Let:
- \( W \) be set of natural numbers
- \( E \) be the expression \( n - k \)

Clearly:
- \( (n - k \geq 0) \) which implies \( n - k \) is in \( W \).
- Can easily prove:
  \[ \begin{align*}
  & (n - k = a) \\
  & k := k + 1; f := f * k \\
  & (n - k = a - 1 < a)
  \end{align*} \]