Programming With Invariants

Complex algorithms are easier to code if we design them around invariants – facts that we know to be true at particular points in execution.

Structured programming constructs, particularly the subset of single-entry, single-exit constructs, make it easier to prove invariants.

Informal invariants are just comments.

Invariants can be formalized using proof rules, which also can be viewed as an axiomatic semantics for the programming language.

Automated theorem provers can be used to help produce precise, formal proofs of program correctness.

Example: FindFirst

Problem: given an array $x[0 \ldots n-1]$ and a value $t$, find the index $p$ of the first element of $x$ such that $x[p] = t$. If there is no such element, set $p = -1$.

Here’s some straightforward code for this problem:

```c
i := 0;
while i < n && x[i] != t do
    i++;
end
if i >= n
    p := -1;
else
    p := i;
```

Using Invariants to Validate Code

Introduce a useful predicate:

$$\text{notin}(q) = (\forall i: 0 \leq i \leq q) \ x[i] \neq t$$

To prove: at the conclusion of the program,

$$(\text{notin}(p - 1) \land x[p] = t) \lor (\text{notin}(n - 1) \land p = -1)$$

Here’s an informally annotated proof:

```c
{ }
i := 0;
{ \text{notin}(i-1) }
while i < n && x[i] != t do
    { \text{notin}(i) }
    i++;
{ \text{notin}(i-1) }
end
{ (i \geq n \lor x[i] = t) \land \text{notin}(i-1) }
if i >= n
    { \text{notin}(n-1) }
p := -1;
{ \text{notin}(n-1) \land p = -1 }
else
    { x[k] = t \land \text{notin}(i-1) }
p := i;
{ x[p] = t \land \text{notin}(p-1) }
{ (\text{notin}(n-1) \land p = -1) \lor (\text{notin}(p-1) \land x[p] = t) }
```
Formalizing Proofs of Correctness
Prove formulas involving assertions about program variables.

The formula

\[
\{ \mathbf{b} \} \mathbf{S} \{ \mathbf{c} \}
\]

says that if the \textbf{precondition} \( \mathbf{b} \) is true before the execution of \( \mathbf{S} \) then the \textbf{postcondition} \( \mathbf{c} \) will be true after the execution of \( \mathbf{S} \).

Note that the formula says nothing about what happens if \( \mathbf{S} \) doesn’t terminate. So proofs involving formulas only prove \textbf{partial correctness}; we must prove facts about termination separately.

Examples:

\[
\{ y \geq 3 \} \mathbf{x} := y + 1 \{ x \geq 4 \}
\]

\[
\{ x + y - c \} \text{while } x > 0 \text{ do } \begin{align*}
    &y := y + 1; \\
    &x := x - 1
\end{align*} \text{end} \{ x + y - c \}
\]

More Proof Rules
Scalar Assignment Axiom

\[
\{ \mathbf{P}[E/x] \} \mathbf{x} := \mathbf{E} \{ \mathbf{P} \}
\]

where \( \mathbf{P}[E/x] \) means \( \mathbf{P} \) with all instances of \( x \) replaced by \( E \).

Array Assignment Axiom

\[
\{ \mathbf{P}[E/a] \} \mathbf{a}[i] := \mathbf{E} \{ \mathbf{P} \}
\]

where \( i \) is a scalar and \( \mathbf{P}[E/a] \) means \( \mathbf{P} \) with all instances of \( a[j] \) replaced by \( \mathbf{E} \).

Consequence Rule

\[
P \Rightarrow P', \{ P' \} \mathbf{S} \{ Q' \}, Q' \Rightarrow Q
\]

\[
\{ P \} \mathbf{S} \{ Q \}
\]

Proof Rules
If we work in a suitably structured language, we can prove the correctness of formulas from a fixed set of axioms and rules of inference, one for each statement type.

Composition Rule

\[
\begin{align*}
\{ \mathbf{P} \} \mathbf{S}_1 \{ \mathbf{Q}_1 \}, \{ \mathbf{Q}_1 \} \mathbf{S}_2 \{ \mathbf{R} \}
\end{align*}
\]

\[
\begin{align*}
\{ \mathbf{P} \} \mathbf{S}_1; \mathbf{S}_2 \{ \mathbf{R} \}
\end{align*}
\]

Conditional Rule

\[
\begin{align*}
\{ \mathbf{P} \land \mathbf{E} \} \mathbf{S}_1 \{ \mathbf{Q}_1 \}, \{ \mathbf{P} \land \lnot \mathbf{E} \} \mathbf{S}_2 \{ \mathbf{Q} \}
\end{align*}
\]

\[
\{ \mathbf{P} \} \text{if } \mathbf{E} \text{ then } \mathbf{S}_1 \text{ else } \mathbf{S}_2 \{ \mathbf{Q} \}
\]

while Rule

\[
\begin{align*}
\{ \mathbf{P} \land \mathbf{E} \} \mathbf{S} \{ \mathbf{P} \}
\end{align*}
\]

\[
\{ \mathbf{P} \} \text{while } \mathbf{E} \text{ do } \mathbf{S} \{ \mathbf{P} \land \lnot \mathbf{E} \}
\]

Proof Tree Example

\[
\begin{align*}
\text{------------------------ (ASSIGN)} \\
\{ x + y - c \} \\
&y := y + 1 \\
&\{ x + y - c + 1 \}
\end{align*} 
\]

\[
\begin{align*}
\text{------------------------ (CONSEQ)} \\
\{ x + y - c \land x > 0 \} \\
&y := y + 1 \\
&x := x - 1 \\
&\{ x + y - c + 1 \}
\end{align*} 
\]

\[
\begin{align*}
\text{------------------------ (COMP)} \\
\{ x + y - c \land x > 0 \} \\
&y := y + 1; x := x - 1 \\
&\{ x + y - c \}
\end{align*} 
\]

\[
\begin{align*}
\text{------------------------ (WHILE)} \\
\{ x + y - c \land x > 0 \} \\
&\text{while } x > 0 \text{ do } y := y + 1; x := x - 1 \text{ end} \\
&\{ x + y - c \land x \leq 0 \}
\end{align*} 
\]

\[
\begin{align*}
\text{------------------------ (CONSEQ)} \\
\{ x + y - c \land x > 0 \} \\
&\text{while } x > 0 \text{ do } y := y + 1; x := x - 1 \text{ end} \\
&\{ x + y - c \}
\end{align*} 
\]

\[
\begin{align*}
\text{------------------------ (WHILE)} \\
\{ x + y - c \land x > 0 \} \\
&\text{while } x > 0 \text{ do } y := y + 1; x := x - 1 \text{ end} \\
&\{ x + y - c \}
\end{align*} 
\]

\[
\begin{align*}
\text{------------------------ (CONSEQ)} \\
\{ x + y - c \land x > 0 \} \\
&\text{while } x > 0 \text{ do } y := y + 1; x := x - 1 \text{ end} \\
&\{ x + y - c \}
\end{align*} 
\]
Axiomatic Semantics

We can also view the proof rules and axioms for a language as an *axiomatic semantics* that defines the language!

Gives a very clean semantics for structured statements, but things get more complicated if we add features like:

- expressions with side-effects
- statements that break out of loops
- procedures
- non-trivial data structures and aliases

Other forms of semantic definition, e.g., *natural semantics*, also use similar *logical* structures.