Optimising Compilers

`while` Language

$$\begin{aligned} C ::= {} & \text{skip} \\ & \mid C_1; C_2 \\ & \mid X := E \\ & \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \\ & \mid \text{while } B \text{ do } C \end{aligned}$$

E are arithmetic expressions, can't contain program statements and don't have side effects. B are boolean expressions.

Hoare Logic

$\{P\}\ C\ \{Q\}$ holds semantically ($\vDash \{P\}\ C\ \{Q\}$) iff C executed in an initial state satisfying P either fails to terminate or terminates with a state satisfying Q.

$\{P\}\ C\ \{Q\}$ **always holds if C doesn't terminate**.

- $\vDash \{\bot\}\ C\ \{Q\}$: says nothing about C as $\bot$ never holds in an initial state.
- $\vDash \{\top\}\ C\ \{Q\}$: says that whenever C halts, Q holds.
- $\vDash \{P\}\ C\ \{\top\}$: holds for any P and C, as $\top$ always holds in the terminal state.
- $\vDash \{P\}\ C\ \{\bot\}$: says that C **doesn't halt**, as $\bot$ can never hold in a terminal state.

Auxiliary Variables are used to refer to initial values of program variables, so we can meaningfully represent changes in variables.

Proof System

$\vdash_{\text{FOL}} P$ holds if we can show P holds in first-order-logic.

$$\frac{}{\vdash \{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\vdash \{P[E/X]\}\ X := E\ \{P\}} \qquad \frac{\vdash \{P\}\ C_1\ \{Q\} \quad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

$$\frac{\vdash \{P \land B\}\ C_1\ \{Q\} \quad \vdash \{P \land \neg B\}\ C_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{Q\}} \qquad \frac{\vdash \{P \land B\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{while}\ B\ \texttt{do}\ C\ \{P \land \neg B\}}$$

$$\frac{\vdash_{\text{FOL}} P_1 \implies P_2 \quad \vdash \{P_2\}\ C\ \{Q_2\} \quad \vdash_{\text{FOL}} Q_2 \implies Q_1}{\vdash \{P_1\}\ C\ \{Q_1\}}$$

Proof Outlines

Rather than writing full trees, it's clearer to write proofs vertically as proof outlines. Annotate each command with the pre- and post- condition of each subcommand. Sequences of assertions indicate use of the rule of consequence, and we omit sides which don't change the assertions, along with the derivations of implications.

Every assertion needs to imply the next, so we need enough information in each one. Generally only an issue when deciding on a loop invariant.

Loop Invariants

The P used in the loop rule:

- Must hold initially.
- Must be preserved by the loop body when B is true.
- Must imply some desired postcondition when B is false.

Can normally work out what it is by looking at what stays the same on each iteration of the loop, and what we can use with $\neg B$ to reach the desired postcondition. Also need to include any information that's used to make the implications in the body make sense.

Verification Condition Generation

Finding loop invariants is difficult. Code that's been annotated with loop invariants (C_a) can be fed to a verification condition generator, along with a precondition P and postcondition Q, to get a set of assertions on program variables $\text{VC}(P, C_a, Q)$. If all the assertions hold, then $\{P\}\ C\ \{Q\}$ holds.

An SMT solver can then produce a program correctness proof using the deduction system above with the assertions, as it can perform the $\vdash_{\text{FOL}}$ proofs.

Dynamic Semantics of `while`

Separation Logic

Model Checking