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.

Dynamic Semantics of `while`

Stacks are **total** functions from program variables to integers ($s \in \text{Stack} = \text{Var} \to \mathbb{Z}$) defining the value of every program variable.

Define $\mathcal{E}[\![E]\!](s)$ and $\mathcal{B}[\![B]\!](s)$ as evaluating the expression in a stack:

$$
\begin{aligned}
\mathcal{E}[\![N]\!](s) &= N \\
\mathcal{E}[\![X]\!](s) &= s(X) \\
\mathcal{E}[\![E_1 + E_2]\!](s) &= \mathcal{E}[\![E_1]\!](s) + \mathcal{E}[\![E_2]\!](s) \\
&\dots
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}[\![\texttt{T}]\!](s) &= \top \\
\mathcal{B}[\![\texttt{F}]\!](s) &= \bot \\
\mathcal{B}[\![E_1 \le E_2]\!](s) &= \begin{cases} \top & \text{, if } \mathcal{E}[\![E_1]\!](s) \le \mathcal{E}[\![E_2]\!](s) \\ \bot & \text{, otherwise} \end{cases} \\
&\dots
\end{aligned}
$$

$$\frac{\mathcal{E}[\![E]\!](s) = N}{\langle X := E, s\rangle \to \langle \texttt{skip}, s[X \mapsto N]\rangle}$$

$$\frac{\mathcal{B}[\![B]\!](s) = \top}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \to \langle C_1, s\rangle} \qquad \frac{\mathcal{B}[\![B]\!](s) = \bot}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \to \langle C_2, s\rangle}$$

$$\dots$$

A configuration can terminate: $\exists s'.\ \langle C, s\rangle \to^* \langle \texttt{skip}, s'\rangle$.

A configuration can fail to terminate: $\langle C, s\rangle \to^\omega$.

Interpretations

Define interpretations of assertions on a specific stack.

- **$[\![P]\!]$ is the set of stacks s which satisfy P.**
- **$[\![t]\!](s)$ is the value of interpreting term t in the stack s.**

$$
\begin{aligned}
[\![\bot]\!] &= \emptyset \\
[\![\top]\!] &= \text{Stack} \\
[\![P \lor Q]\!] &= [\![P]\!] \cup [\![Q]\!] \\
[\![P \land Q]\!] &= [\![P]\!] \cap [\![Q]\!] \\
[\![t_1 = t_2]\!] &= \{s \in \text{Stack} \mid [\![t_1]\!](s) = [\![t_2]\!](s)\} \\
[\![p(t_1, ..., t_n)]\!] &= \{s \in \text{Stack} \mid [\![p]\!]([\![t_1]\!](s), ..., [\![t_n]\!](s))\} \\
[\![\forall x.\ P]\!] &= \{s \in \text{Stack} \mid \forall N.\ s[x \mapsto N] \in [\![P]\!]\} \\
[\![\exists x.\ P]\!] &= \{s \in \text{Stack} \mid \exists N.\ s[x \mapsto N] \in [\![P]\!]\}
\end{aligned}
$$

$$
\begin{aligned}
[\![B]\!] &= \{s \in \text{Stack} \mid \mathcal{B}[\![B]\!](s) = \top\} \\
[\![f(t_1, ..., t_n)]\!](s) &= [\![f]\!]([\![t_1]\!](s), ..., [\![t_n]\!](s))
\end{aligned}
$$

Can now define:

$$\vDash \{P\}\ C\ \{Q\} \overset{\text{def}}{=} \forall s, s'.\ (s \in [\![P]\!]) \land (\langle C, s\rangle \to^* \langle \texttt{skip}, s'\rangle) \implies s' \in [\![Q]\!]$$

$\texttt{while}_p$ Language

`while` with pointers. Introduce new commands:

$$\begin{aligned} C ::= \; & X := [E] \\ & \mid [E_1] := E_2 \\ & \mid X := \texttt{alloc}(E_0, ..., E_n) \\ & \mid \texttt{dispose}(E) \end{aligned}$$

Commands are now evaluated with a heap and a stack: $h \in \text{Heap} = (\text{Loc} \setminus \{\texttt{null}\}) \to \mathbb{Z}$, $l \in \text{Loc} = \mathbb{N}$. The existing commands can be easily extended to handle the extra state.

The new commands **fail** ($\lightning$) if the given locations aren't allocated, but otherwise have straightforward dynamic semantics. Allocating memory needs to find a contiguous block of unallocated memory to place the values in.

$$\frac{\mathcal{E}[\![E]\!](s) = l \quad l \in \text{dom}(h) \quad h(l) = N}{\langle X := [E], s, h\rangle \to \langle \texttt{skip}, s[X \mapsto N], h\rangle} \qquad \frac{\mathcal{E}[\![E]\!](s) = l \quad l \notin \text{Loc}}{\langle X := [E], s, h\rangle \to \lightning} \qquad \frac{\mathcal{E}[\![E]\!](s) = l \quad l \notin \text{dom}(h)}{\langle X := [E], s, h\rangle \to \lightning}$$

$$\frac{\mathcal{E}[\![E_1]\!](s) = l \quad l \in \text{dom}(h) \quad \mathcal{E}[\![E_2]\!] = N}{\langle [E_1] := E_2, s, h\rangle \to \langle \texttt{skip}, s, h[l \mapsto N]\rangle} \qquad \frac{\mathcal{E}[\![E_1]\!](s) = l \quad l \notin \text{Loc}}{\langle [E_1] := E_2, s, h\rangle \to \lightning} \qquad \frac{\mathcal{E}[\![E_1]\!](s) = l \quad l \notin \text{dom}(h)}{\langle [E_1] := E_2, s, h\rangle \to \lightning}$$

$$\frac{\mathcal{E}[\![E]\!](s) = l \quad l \in \text{dom}(h) \quad h(l) = N}{\langle \texttt{dispose}(E), s, h\rangle \to \langle \texttt{skip}, s, h \setminus \{(l, N)\}\rangle} \qquad \frac{\mathcal{E}[\![E]\!](s) = l \quad l \notin \text{Loc}}{\langle \texttt{dispose}(E), s, h\rangle \to \lightning} \qquad \frac{\mathcal{E}[\![E]\!](s) = l \quad l \notin \text{dom}(h)}{\langle \texttt{dispose}(E), s, h\rangle \to \lightning}$$

$$\frac{\mathcal{E}[\![E_0]\!](s) = N_0 \quad ... \quad \mathcal{E}[\![E_n]\!](s) = N_n \quad \forall i \in 0, ..., n.\ l + i \notin \text{dom}(h) \quad l \neq \texttt{null}}{\langle X := \texttt{alloc}(E_0, ..., E_n), s, h\rangle \to \langle \texttt{skip}, s[X \mapsto l], h[l \mapsto N_0, ..., l + n \mapsto N_n]\rangle}$$

Allocation introduces non-determinism, as we can use the allocated locations to affect program behaviour.

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.

Properties of Hoare Logic

Soundness: If $\vdash \{P\}\ C\ \{Q\}$ then $\vDash \{P\}\ C\ \{Q\}$.
Any triple derivable using the syntactic proof system also holds semantically.

~~Completeness~~: If $\vDash \{P\}\ C\ \{Q\}$ then $\vdash \{P\}\ C\ \{Q\}$.
The converse of Soundness, Hoare logic **isn't complete** as it inherits incompleteness from arithmetic.

~~Decidability~~: There's a computable function f such that $f(P, C, Q) = \top \iff \{P\}\ C\ \{Q\}$.
We can encode Turing machines in the while language, so as the Halting Problem is undecidable, so is Hoare logic.

Separation Logic

Extend Hoare Logic to deal with heaps. The **frame rule** combined with the $C_1; C_2$ rule allows us to verify independent subprograms which use distinct program variables. Note that $\text{mod}(C)$ gives the modified **program variables**, nothing to do with the heap.

$$\frac{\vdash \{P\}\ C\ \{Q\} \quad \text{mod}(C) \cap \text{fv}(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

Introduce a new **assertion** $t_1 \mapsto t_2$, which asserts that heap location t_1 maps to the value t_2 and **also asserts ownership** of heap location t_1. It's the same syntax as use in the stack and heap, but it's an assertion.

Introduce the separating conjunction $*$, where $P * Q$ asserts **that** P **and** Q **hold** (like $P \wedge Q$) and further that **the heap locations owned by** P **and** Q **are disjoint**. The neutral element for $*$ is emp.

Assertions should now **always** contain a heap-describing part that **fully describes the heap**, and a standard propositional part. Eg. $\text{emp} \wedge P \wedge (Q \vee R)$ and $(X \mapsto Y * Z \mapsto Y * Q \mapsto W) \wedge P \wedge (Q \vee R)$.

$t_1 \mapsto t_2 \wedge t_3 \mapsto t_4$ means that both $t_1 \mapsto t_2$ and $t_3 \mapsto t_4$ fully describe the heap, so $t_1 = t_3$ and by consequence $t_2 = t_4$.

$t \mapsto X, Y, Z$ is shorthand for $(t \mapsto X) * (t + 1 \mapsto Y) * (t + 2 \mapsto Z)$.

Under separation logic, $\{P\}\ C\ \{Q\}$ means that:

- C doesn't fault when evaluated from an initial state satisfying P and
- **For any** h_1 **satisfying** P**, if** C **is evaluated from an initial state with heap** $h_1 \uplus h_F$ **and** C **terminates, then the final state has heap** $h_1' \uplus h_F$ **where** h_1' **satisfies** Q.

The second condition strengthens the version from Hoare Logic to enforce that triples have to satisfy framing, **by requiring that they preserve any disjoint heaps** h_F.

* Proof System

$$\frac{\vdash \{P\}\ C\ \{Q\} \quad \text{mod}(C) \cap \text{FV}(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The frame rule allows composition of subproofs for programs which operate on disjoint subsets of the heap.

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{\exists x.\ P\}\ C\ \{\exists x.\ Q\}}$$

The structural $\exists$ rule allows eliminating existentials that we can't provide a witness for.

$$\frac{}{\vdash \{\exists x.\ E_1 \mapsto x\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}}$$

The simpler assignment rule just remaps E_1 to point to E_2, nothing special.

$$\frac{}{\vdash \{\exists x, v.\ X = x \wedge E \mapsto v\}\ X := [E]\ \{X = v \wedge E[x/X] \mapsto v\}}$$

The more complicated assignment rule updates program variable X but also has to ensure that E isn't changed by the update.

$$\frac{}{\vdash \{X = x \wedge \mathrm{emp}\}\ X := \texttt{alloc}(E_0, ..., E_n)\ \{X \mapsto E_0[x/X], ..., E_n[x/X]\}}$$

Allocation introduces a new $\mapsto$ assertion for every allocated expression.

$$\frac{}{\vdash \{E \mapsto t\}\ \texttt{dispose}(E)\ \{\mathrm{emp}\}}$$

Deallocation just empties the heap.

$\exists$ is a friend: introduce whenever there's a new variable that's not mentioned in the pre/postconditions, then eliminate it using $\vdash_{\mathrm{FOL}}$ when we have a suitable concrete variable, or use the $\exists$ structural rule to eliminate it from both sides.

Check this is actually how we deal with $\exists$.

Model Checking