Hoare Logic and Model Checking

`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:

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

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

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

...

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.**

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

Can now define:

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

$\texttt{while}_p$ 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.

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

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

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

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 generalising axioms that deal with fixed auxiliary variables (those below) to ones wrapped in $\exists$. This is extremely important as separation logic should end up with lots of quantified variables for the various unknown values on the heap.
The usage is 'axiom stacked on exists rule stacked on consequence rule', where the implications transform into and out of the quantified versions.

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

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

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

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

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

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

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

Deallocation just empties the heap.

* Proof Outlines

Don't need to show all the complicated uses of the frame/structural rules, can just give a high-level outline using the rule of consequence. Still need to make sure that the assertions given logically follow.

Introduce exists-bound variable for *any* variable not already existing in the pre/postcondition, regardless of whether it's an auxiliary variable or a mathematical variable, such as $\exists y, \alpha, \beta.\ \text{list}(y, a :: b)$.

The $X := [E]$ rule will eliminate an $\exists$-bound variable α (assuming $E \mapsto \alpha$), as it should introduce an $X = \alpha$ assertion which can be used to trivially derive a version of the assertion without the $\exists\alpha$, replacing α with E throughout.

Using mathematical operations in assertions is fine: $(X = Y \wedge \alpha = \beta :: \gamma) * \text{list}(Z, \alpha +\!\!+ \alpha)$.

Example proof outline for:

$\{\text{list}(X, \alpha) \wedge X \neq \texttt{null}\}$
$E := [X];\ P := [X + 1];\ Y := \texttt{alloc}(E, P);\ \texttt{dispose}(X);\ \texttt{dispose}(X + 1)$
$\{\text{list}(Y, \alpha) \wedge Y \neq \texttt{null}\}$

$\{\text{list}(X, \alpha) \wedge X \neq \texttt{null}\}$
$\{\exists y, \beta, \gamma.\ (\text{list}(y, \gamma) * X \mapsto \beta, y) \wedge \alpha = \beta :: \gamma\}$
$E := [X];$
$\{\exists y, \beta, \gamma.\ (\text{list}(y, \gamma) * X \mapsto \beta, y) \wedge \alpha = \beta :: \gamma \wedge E = \beta\}$
$\{\exists y, \gamma.\ (\text{list}(y, \gamma) * X \mapsto E, y) \wedge \alpha = E :: \gamma\}$ Note that we removed $\exists\beta$ using $E = \beta$.
$P := [X + 1];$
$\{\exists \gamma.\ (\text{list}(P, \gamma) * X \mapsto E, P) \wedge \alpha = E :: \gamma\}$
$Y := \texttt{alloc}(E, P)$
$\{\exists \gamma.\ (\text{list}(P, \gamma) * X \mapsto E, P * Y \mapsto E, P) \wedge \alpha = E :: \gamma\}$
$\{(\text{list}(Y, \alpha) * X \mapsto E, P) \wedge Y \neq \texttt{null}\}$ Repacking with the definition of $\text{list}(Y, E :: \gamma)$.
$\texttt{dispose}(X);\ \texttt{dispose}(X + 1);$
$\{\text{list}(Y, \alpha) \wedge Y \neq \texttt{null}\}$

Loop Invariants

Loop invariants are more complicated in $\texttt{while}_p$ as the invariant **needs to carry enough information to fully specify the heap on each iteration**.

Datatypes

Lists

The list predicate $\text{list}(t, \alpha$ associates a program variable t with the mathematical representation of the list it describes α:

$\text{list}(t, []) = \text{emp} \wedge t = \texttt{null}$
$\text{list}(t, \beta :: \gamma) = \exists y.\ (t \mapsto \beta, y) * \text{list}(y, \gamma)$

Partial Lists

Partial lists $\text{plist}(t_1, \alpha, t_2)$ represent a prefix of a list where t_1 points to a list representing α with tail pointer t_2:

$\text{plist}(t_1, [], t_2) = \text{emp} \wedge t_1 = t_2$

$\text{plist}(t_1, \beta :: \gamma, t_2) = \exists y.\ (t_1 \mapsto \beta, y) * \text{plist}(y, \gamma, t_2)$

Useful for describing sections of lists: $\text{list}(t, \beta +\!\!+ \gamma) \iff \exists y.\ \text{plist}(t, \beta, y) * \text{list}(y, \gamma)$.

Circular Lists

$\text{clist}(X, \alpha) = \text{plist}(X, \alpha, X)$

* Total Correctness

A total correctness triple strengthens partial correctness triples to further assert that the command halts.

$$\vDash [P]\ C\ [Q] \equiv \forall s.\ s \in [\![P]\!] \implies \left[\neg\langle C, s\rangle \to^{\omega}\right] \wedge \left[\forall s'.\ \langle C, s\rangle \to^{*} \langle \texttt{skip}, s'\rangle \implies s' \in [\![Q]\!]\right]$$

All normal proof rules are the same except for the loop rule:

$$\frac{\vdash \{P \wedge B \wedge (\boldsymbol{t = n})\}\ C\ \{P \wedge (\boldsymbol{t < n})\} \quad \vdash_{\text{FOL}} \boldsymbol{P \wedge B \implies t \geq 0}}{\vdash \{P\}\ \texttt{while}\ B\ \texttt{do}\ C\ \{P \wedge \neg B\}}$$

The **variant** t is a term that decreases from some start auxiliary variable n and is non-negative. The variant can be any term, it doesn't have to occur in the command.

Model Checking

Model checking is exhaustive search for solutions: automatic and easier than Hoare logic, but requires finite models.

If a model fails to verify a proposition, it can report why: eg. which path through the state space failed to satisfy it. Means we can prove things by asking for a proof of a refutation.

Kripke Structures

A finite Kripke structure is a tuple (S, S_0, R, L):

- S is a finite set of states.
- $S_0 \subseteq S$ is a set of start states.
- $R : S \to S$ is the transition **relation**.
- $L : S \to \mathcal{P}(AP)$ is the labelling function specifying what properties hold in which states.

AP are a given set of atomic properties. We can model finite state automata by including inputs as possible state.

A **path** in the transition system (S, R) is a function $\pi : \mathbb{N} \to S$:

- $\pi(i)$ is the i^{th} element of π (first element is $\pi(0)$).
- $\pi \downarrow i$ is $\texttt{drop i}\ \pi$, the i^{th} tail of π.

Path $R\ s\ \pi$ defines that π is a path from s, and is true iff $\pi(0) = s \wedge \forall i.\ R\ (\pi(i))\ (\pi(i+1))$.

Symbolic Model Checking

Some models are too big, like a system with n flip-flops has 2^n states which gets intractable fast. Can instead represent the system as operations on BDDs, states as specific tuples. Generally more efficient.

Linear Temporal Logic (LTL)

$M = (S, S_0, R, L) \vDash \phi$ means that ϕ holds on all R-paths starting from a member of S_0:

$$\forall \pi, s.\ (s \in S_0 \wedge \text{Path}\ R\ s\ \pi) \implies [\![\phi]\!]_M(\pi)$$

F and G talk about **tails of the path**, not new paths: we're given a fixed path at the start and just see different suffixes of it.

$[\![\phi]\!]_M(\pi)$ is defined as:

$\phi ::= p$	Atomic proposition	$[\![p]\!]_M(\pi)$	$= p \in L(\pi(0))$
$\mid \neg\phi$	Negation	$[\![\neg\phi]\!]_M(\pi)$	$= \neg([\![\phi]\!]_M(\pi))$
$\mid \phi_1 \vee \phi_2$	Disjunction	$[\![\phi_1 \vee \phi_2]\!]_M(\pi)$	$= [\![\phi_1]\!]_M(\pi) \vee [\![\phi_2]\!]_M(\pi)$
$\mid ...$			...
$\mid X\phi$	Successor	$[\![X\phi]\!]_M(\pi)$	$= [\![\phi]\!]_M(\pi \downarrow 1)$
$\mid F\phi$	Sometimes	$[\![F\phi]\!]_M(\pi)$	$= \exists i \geq 0.\ [\![\phi]\!]_M(\pi \downarrow i)$
$\mid G\phi$	Always	$[\![G\phi]\!]_M(\pi)$	$= \forall i \geq 0.\ [\![\phi]\!]_M(\pi \downarrow i)$
$\mid [\phi_1\ U\ \phi_2]$	Until	$[\![[\phi_1\ U\ \phi_2]]\!]_M(\pi)$	$= \exists i \geq 0.\ \left[[\![\phi_2]\!]_M(\pi \downarrow i)\right] \wedge \left[\forall j < i.\ [\![\phi_1]\!]_M(\pi \downarrow j)\right]$

LTL quantifies over **all paths**: the formula must hold on **any path** through the system.

- $G(F\phi)$: ϕ holds **infinitely often** along every path (at each state on every path, somewhere further along the path there's a state in which ϕ holds).
- $F(G\phi)$: ϕ eventually ϕ holds permanently (on all paths, there's a point after which ϕ holds for all states).
- $G(\phi \implies F\psi)$: if ϕ holds in any state in the path, then there is some state further along in which ψ holds.
- $G(\phi \implies [\phi\ U\ \psi])$: if ϕ holds at any state in the path, then it will continue to hold at least until ψ holds.

Computation Tree Logic (CTL)

CTL expresses properties of **states**, and reasons about possible paths (all paths from here, some paths from here, ...). Doesn't really reason about trees, we just consider paths through subtrees.

Define $(S, S_0, R, L) \vDash \psi$ as $\forall s \in S_0.\ [\![\psi]\!]_M(s)$. LTL considers all paths, CTL just starts us off at the start states.

$\psi ::= p$	Atomic proposition	$[\![p]\!]_M(s)$	$= p \in L(s)$
$\mid \neg\psi$	Negation	$[\![\neg\psi]\!]_M(s)$	$= \neg([\![\psi]\!]_M(s))$
$\mid \psi_1 \vee \psi_2$	Disjunction	$[\![\psi_1 \vee \psi_2]\!]_M(s)$	$= [\![\psi_1]\!]_M(s) \vee [\![\psi_2]\!]_M(s)$
$\mid ...$			...
$\mid AX\psi$	All successors	$[\![AX\psi]\!]_M(s)$	$= \forall s'.\ R\ s\ s' \implies [\![\psi]\!]_M(s')$
$\mid EX\psi$	Some successors	$[\![EX\psi]\!]_M(s)$	$= \exists s'.\ R\ s\ s' \wedge [\![\psi]\!]_M(s')$
$\mid A[\psi_1\ U\ \psi_2]$	Until, along all paths	$[\![A[\psi_1\ U\ \psi_2]]\!]_M(s)$	$= \forall \pi.\ \text{Path}\ R\ s\ \pi \implies \exists i \geq 0.\ [[\![\psi_2]\!]_M(\pi(i))] \wedge [\forall j < i.\ [\![\psi_1]\!]_M(\pi(j))]$
$\mid E[\psi_1\ U\ \psi_2]$	Until, along some paths	$[\![E[\psi_1\ U\ \psi_2]]\!]_M(s)$	$= \exists \pi.\ \text{Path}\ R\ s\ \pi \implies$ — · · —
$\mid AF\psi = A[\mathtt{T}\ U\ \psi]$		somewhere on all paths from s.	
$\mid EF\psi = E[\mathtt{T}\ U\ \psi]$		somewhere on some path from s.	
$\mid AG\psi = \neg EF(\neg\psi)$		always on all paths from s.	
$\mid EG\psi = \neg AF(\neg\psi)$		always on some path from s.	
$\mid \mathbf{A}[\psi_\mathbf{1}\ \mathbf{W}\ \psi_\mathbf{2}]$		$= \neg\mathbf{E}[(\psi_\mathbf{1} \wedge \neg\psi_\mathbf{2})\ \mathbf{U}\ (\neg\psi_\mathbf{1} \wedge \neg\psi_\mathbf{2})]$	(Unless, along all paths)
$\mid \mathbf{E}[\psi_\mathbf{1}\ \mathbf{W}\ \psi_\mathbf{2}]$		$= \neg\mathbf{A}[(\psi_\mathbf{1} \wedge \neg\psi_\mathbf{2})\ \mathbf{U}\ (\neg\psi_\mathbf{1} \wedge \neg\psi_\mathbf{2})]$	(Unless, along some paths)

$A[\psi_1\ W\ \psi_2]$ and $E[\psi_1\ W\ \psi_2]$ are the partial-correctness versions of $A[\psi_1\ U\ \psi_2]$ and $E[\psi_1\ U\ \psi_2]$: they hold if either:

- ψ_1 always holds along the path (an 'infinite loop'), or if
- ψ_1 holds up to a finite point then ψ_2 holds after that point.

Examples:

- $EF(\phi \wedge \psi)$: it's possible to reach a state where $\phi \wedge \psi$ holds (there exists a path where at some point $\phi \wedge \psi$).
- $AG(\phi \implies AF\psi)$: if ϕ holds in any state, then at some point in the future ψ will hold.
- $AG(AF\phi)$: from any state, you can reach a state where ϕ holds, so ϕ holds infinitely often along every path.
- $AG(\text{Req} \implies AX(A[\neg\text{Req } U \text{ Ack}]))$: If Req holds in any state, then at some point later Ack will hold, and until that point Req will be false. The AX isn't necessary.

Model Checking

For LTL, consider every path defined by R from S_0 and just check the formula holds using the semantic definition.

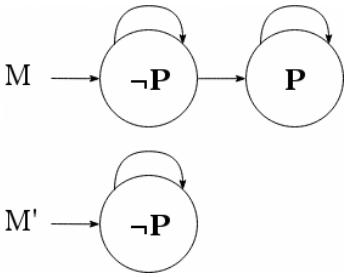
For CTL, compute $F(\psi) = \{s \mid [\![\psi]\!]_M(s)\}$ using top-down dynamic programming (just recursion with caching), then check that $S_0 \subseteq F(\psi)$. Symbolic model checking represents the sets as BDDs.

- $F(p) = \{s \mid p \in L(s)\}$
- $F(AX\psi) = \{s \mid \forall s'.\ R\ s\ s' \wedge s' \in F(\psi)\}$: the states where all successors satisfy the subformula.
- $F(EX\psi) = \{s \mid \exists s'.\ R\ s\ s' \wedge s' \in F(\psi)\}$: the states where at least one successor satisfies the subformula.
- $F(E[\psi_1\ U\ \psi_2])$: mark all states in $F(\psi_2)$, then mark all states in $F(\psi_1)$ with at least one successor that's marked, and repeat until we reach a fixed point.
- $F(A[\psi_1\ U\ \psi_2])$: same as above but require all successors to be marked.
- ...

LTL vs CTL

LTL can't, CTL can

Every path in M is also a path in M', so if $M \vDash \phi$ then $M' \vDash \phi$.
However, $\phi \equiv$ **"can always reach a state satisfying P"** holds in M but not M', so there's a contradiction.
LTL can't express that **there exists a path**, it can only reason about all or some paths that exist.



CTL would express this as $AG(EF\ \text{P})$: from any state, there's always a path to a state where P holds.

CTL can't, LTL can

'On every path there is a point after which p is always true **on that path**'.

Want something like $AF\psi$ where ψ is 'p is true from this state on', but in CTL ψ has to start with a path quantifier A or E, which talks about **all or some paths rather than the current path**.

LTL can express this as $F(G\,p)$.

CTL*

Combine LTL and CTL: can describe path formulae and state formulae:

$$
\begin{array}{lll}
\phi ::= & \psi & \\
& \mid \neg\phi & \text{Negation} \\
& \mid \phi_1 \vee \phi_2 & \text{Disjunction} \\
& \mid \ldots & \\
& \mid X\phi & \text{Successor} \\
& \mid F\phi & \text{Sometimes} \\
& \mid G\phi & \text{Always} \\
& \mid [\phi_1\ U\ \phi_2] & \text{Until}
\end{array}
$$

$$
\begin{array}{lll}
\psi ::= & p & \text{Atomic formula} \\
& \mid \neg\psi & \text{Negation} \\
& \mid \psi_1 \vee \psi_2 & \text{Disjunction} \\
& \mid \ldots & \\
& \mid A\phi & \text{All paths} \\
& \mid E\phi & \text{Some paths}
\end{array}
$$

Defined mutually recursively between state and path formulae. Ensures that the connectives line up in the right orders.

State formulae ψ are true for a state s like CTL, path formulae are true of a path π like LTL.

For CTL*, $(S, S_0, R, L) \vDash \psi$ **is defined as** $\forall s \in S_0.\ [\![\psi]\!]_M(s)$ **(the same as in CTL).**

All LTL formulae can be proved by quantifying them with A (to match up the definition of $\vDash$): $M \vDash_{\text{LTL}} \phi \iff M \vDash_{\text{CTL}^*} A\phi$.

Simulation and Abstraction

Simulation

Let M_1 and M_2 be two models, with transition relations R_1 and R_2.

$H : S_1 \to S_2 \to \mathbb{B}$ is a **simulation relation between** R_1 **and** R_2 if for every step of R_1 there is a corresponding step of R_2:

$$\forall s_1, s_1' \in S_1, s_2 \in S_2.\ H\ s_1\ s_2 \wedge R_1\ s_1\ s_1' \implies \exists s_2' \in S_2.\ R_2\ s_2\ s_2' \wedge H\ s_1'\ s_2'$$

If there's a simulation relation between M_1 and M_2, then M_2 **simulates** M_1. A state s_2 in M_2 simulates a state s_1 in M_1 iff $H\ s_1\ s_2$.

Also interested in a simulation relation H_{AP} between the abstract properties of the two machines, but not detailed.

Bisimulation

$H : S_1 \to S_2 \to \mathbb{B}$ is a **bisimulation relation between** R_1 **and** R_2 if for every step of R_1 there is a corresponding step of R_2, and vice-versa (if M_1 simulates M_2 and M_2 simulates M_1):

- $\forall s_1, s_1' \in S_1, s_2 \in S_2.\ H\ s_1\ s_2 \wedge R_1\ s_1\ s_1' \implies \exists s_2' \in S_2.\ R_2\ s_2\ s_2' \wedge H\ s_1'\ s_2'$
- $\forall s_1 \in S_1, s_2, s_2' \in S_2.\ H\ s_1\ s_2 \wedge R_2\ s_2\ s_2' \implies \exists s_1' \in S_1.\ R_1\ s_1\ s_1' \wedge H\ s_1'\ s_2'$

Bisimulation Equivalence

$M \equiv M'$ iff:

- there's a bisimulation relation B between R and R'.
- $\forall s_0 \in S_0.\ \exists s_0' \in S_0'.\ B s_0 s_0'$.
- $\forall s_0' \in S_0'.\ \exists s_0 \in S_0.\ B s_0 s_0'$.
- $\forall s \in S_0, s' \in S_0'.\ B\ s\ s' \implies L(s) = L'(s')$.

If $M_1 \equiv M_2$ then for any CTL* formula ψ, $M_1 \vDash \psi \iff M_2 \vDash \psi$.

Abstraction

An abstraction simplifies a model: several concrete states and properties may be merged, and an abstract path may represent several concrete paths.

$M \preceq \overline{M}$ means $\overline{M}$ is an abstraction of M:

- $\overline{M}$ simulates M with relation H and abstract property relation H_{AP}.
- $\forall s_0 \in S_0.\ \exists \overline{s_0} \in \overline{S_0}.\ H\ s_0\ \overline{s_0}$ (all start states in M have an equivalent start state in $\overline{M}$).
- $\forall s \in S, \overline{s} \in \overline{S}.\ H\ s\ \overline{s} \implies H_{AP}\ L(s)\ \overline{L}(\overline{s})$ (if a state simulates another, then the abstract properties are also abstracted).

Intuitively, read $M_1 \preceq M_2$ as 'the states of M_1 are a subset of the states of M_2' (plus the other conditions): M_2 clusters some states together.

ACTL

Define a subset ACTL of CTL without E-properties and with negation only of atomic properties p, $p_1 \wedge p_2$, ...

If $M \preceq \overline{M}$ then for any ACTL state formula ψ, $\overline{M} \vDash \psi \implies M \vDash \psi$. Strictly an implication, the converse doesn't hold. Formalisation of an abstract interpretation from Optimising Compilers.