Invariant Generation

CS560: Reasoning About Programs

Roopsha Samanta
Roadmap

Previously

- Semi-automating Hoare logic using verification condition generation

Today

- Automated invariant generation
- Forward propagation using strongest postconditions
- Abstract interpretation
Invariant vs. Inductive Invariant

- A formula $I$ annotating location $L$ of a program $S$ is an invariant iff it holds whenever control reaches location $L$:
  
  for each computation $s_0, s_1, s_2, ...$ of $S$ and each index $i$, $pc(s_i) = L$ implies $s_i \models I$.

- The annotations of $S$ are invariant iff each annotation is invariant.

- The annotations of $S$ are inductive (or, inductive invariants) iff all corresponding verification conditions are valid.

Hard to prove an annotation is an invariant as there can be an infinite no. of computations.
Invariant vs. Inductive Invariant

- A formula $I$ annotating location $L$ of a program $S$ is an invariant iff it holds whenever control reaches location $L$:
  
  for each computation $s_0, s_1, s_2, \ldots$ of $S$ and each index $i$, $pc(s_i) = L$ implies $s_i \models I$.

- The annotations of $S$ are invariant iff each annotation is invariant.

- The annotations of $S$ are inductive (or, inductive invariants) iff all corresponding verification conditions are valid.

Hard to prove an annotation is an invariant as there can be an infinite no. of computations.

Inductive invariants are the invariants we can prove.
Loop Invariant

- A loop invariant $I$ is an annotation for a loop that must hold at the beginning of every iteration.
  - $I$ must hold before the loop begins
  - $I$ must hold after every iteration
Verification
Condition
Generator

Verification condition
(FOL formula)

Automatic
Theorem prover
(SMT Solver)

Program
Loop invariants
Precondition + Postcondition

Valid ✓
Not valid ✗
Verification
Condition
Generator

Program

Precondition + Postcondition

Invariant Generator

Verification Condition Generator

Automatic Theorem prover (SMT Solver)

Valid ✓

Not valid ✗
Inductive assertion method

- Represent program as control-flow graph with annotations/inductive assertions
- Identify basic paths
- For each basic path: check if corresponding Hoare triple is valid

VCs

{pre} $x := 0; y := 1$ {l}
{l} assume $y > n$ {post}
l skip {l}
{l} assume $y \leq n$; assume $z \neq 0$; $x := x + y; y := y + 1$ {l}
l assume $y \leq n$; assume $z = 0$; $x := x + 1; y := y + 1$ {l}

$\mathcal{L} = \{L_0, L_1, L_2, L_3\}$
$succ(L_0) = \{L_2, L_3\}$
$succ(L_1) = \{L_2, L_3\}$
$succ(L_2) = \{L_3\}$
A cutset $\mathcal{L}$ is a set of locations (called cutpoints) such that every path between adjacent cutpoints is a basic path.

**Invariant Generation:**

Find a map $\mu: \mathcal{L} \mapsto \text{FOL}$ such that for each basic path, its corresponding Hoare triple is valid.

The map $\mu$ is called an *inductive assertions map*.
ForwardPropagate(pre, L)

\[ W := \{L_{init}\} \]

\[ \mu(L_{init}) := \text{pre} \]

foreach \( L \in L \setminus \{L_{init}\} \) do \( \mu(L) := \bot \)

while \( W \neq \emptyset \) do

  pick \( L_j \in W \); \( W := W \setminus \{L_j\} \)

  foreach \( L_k \in \text{succ}(L_j) \) do

    if \( \sim (sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \)

    then

      \[ \mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j)) \]

      \[ W := W \cup \{L_k\} \]

return \( \mu \)
ForwardPropagate(pre, L)

\[ W := \{L_{init}\} \]

\[ \mu(L_{init}) := \text{pre} \]

\textbf{foreach} \( L \in L \setminus \{L_{init}\} \) \textbf{do} \( \mu(L) := \bot \)

\textbf{while} \( W \neq \emptyset \) \textbf{do}

\textbf{pick} \( L_j \in W; \ W := W \setminus \{L_j\} \)

\textbf{foreach} \( L_k \in \text{succ}(L_j) \) \textbf{do}

\textbf{if} \( \neg (sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \)

\textbf{then}

\[ \mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j)) \]

\[ W := W \cup \{L_k\} \]

\textbf{return} \( \mu \)

Set of locations in \( L \) that need to be processed
ForwardPropagate\(pre, L\)

\[W := \{L_{init}\}\]

\[\mu(L_{init}) := pre\]

\[
\text{foreach } L \in L\backslash\{L_{init}\} \text{ do } \mu(L) := \bot
\]

\[\text{while } W \neq \emptyset \text{ do}\]

\[\text{pick } L_j \in W; W := W\backslash\{L_j\}\]

\[
\text{foreach } L_k \in \text{succ}(L_j) \text{ do }
\]

\[
\text{if } \neg \left( sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k) \right) \text{ then }
\]

\[
\mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j))
\]

\[W := W \cup \{L_k\}\]

\[\text{return } \mu\]
ForwardPropagate(pre, ℒ)

\[ W := \{ L_{init} \} \]

\[ \mu(L_{init}) := \text{pre} \]

foreach \( L \in \mathcal{L} \setminus \{ L_{init} \} \) do \( \mu(L) := \perp \)

while \( W \neq \emptyset \) do

  pick \( L_j \in W; W := W \setminus \{ L_j \} \)

  foreach \( L_k \in \text{succ}(L_j) \) do

    if \( \neg (sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \)

    then

      \[ \mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j)) \]

      \[ W := W \cup \{ L_k \} \]

return \( \mu \)
ForwardPropagate\((\text{pre, } \mathcal{L})\)

\[ W := \{L_{\text{init}}\} \]

\[ \mu(L_{\text{init}}) := \text{pre} \]

\[
\text{foreach } L \in \mathcal{L} \setminus \{L_{\text{init}}\} \text{ do } \mu(L) := \bot
\]

\[
\text{while } W \neq \emptyset \text{ do }
\]

\[
\text{pick } L_j \in W; W := W \setminus \{L_j\}
\]

\[
\text{foreach } L_k \in \text{succ}(L_j) \text{ do }
\]

\[
\text{if } \neg (sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k))
\]

\[
\text{then }
\]

\[
\mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j))
\]

\[
W := W \cup \{L_k\}
\]

\[
\text{return } \mu
\]
ForwardPropagate(pre, ℒ)

\[ W := \{L_{init}\} \]

\[ \mu(L_{init}) := pre \]

\[ \text{foreach } L \in ℒ\setminus\{L_{init}\} \text{ do } \mu(L) := \bot \]

\[ \text{while } W \neq \emptyset \text{ do} \]

\[ \text{pick } L_j \in W; W := W\setminus\{L_j\} \]

\[ \text{foreach } L_k \in \text{succ}(L_j) \text{ do} \]

\[ \text{if } \neg (sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \]

\[ \text{then} \]

\[ \mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j)) \]

\[ W := W \cup \{L_k\} \]

return \( \mu \)

Undecidable for FOL
ForwardPropagate(pre, \( \mathcal{L} \))

\[
W := \{ L_{\text{init}} \} \\
\mu(L_{\text{init}}) := \text{pre} \\
foreach L \in \mathcal{L} \setminus \{ L_{\text{init}} \} \text{ do } \mu(L) := \bot \\
\text{while } W \neq \emptyset \text{ do} \\
\text{pick } L_j \in W; W := W \setminus \{ L_j \} \\
foreach L_k \in \text{succ}(L_j) \text{ do} \\
\text{if } \neg (sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \\
\text{then} \\
\mu(L_k) := \mu(L_k) \lor sp(S_j; \ldots; S_k, \mu(L_j)) \\
W := W \cup \{ L_k \} \\
\text{return } \mu
\]
Pre: $i = 0 \land n > 0$

while $i < n$

\[ i = i + 1 \]

\[ \forall i = 0 \land n \geq 0 \]

\[ \forall i = 1 \land n > 0 \]

\[ \forall i = 2 \land n > 1 \]

\[ \forall i = k \land n > k \]

\[ \forall i = n \]

Doesn't converge!

But "$0 \leq i \leq n$" is a loop invariant!
ForwardPropagate(pre, ℒ)

\[ W := \{ L_{init} \} \]
\[ \mu(L_{init}) := pre \]

\[
\text{foreach } L \in \mathcal{L}\backslash\{L_{init}\} \text{ do } \mu(L) := \bot
\]

\[ \text{while } W \neq \emptyset \text{ do} \]
\[ \text{pick } L_j \in W; \ W := W\backslash\{L_j\} \]
\[ \text{foreach } L_k \in \text{succ}(L_j) \text{ do} \]
\[ \text{if } \neg(\text{sp}(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \]
\[ \text{then} \]
\[ \mu(L_k) := \mu(L_k) \lor \text{sp}(S_j; \ldots; S_k, \mu(L_j)) \]
\[ W := W \cup \{L_k\} \]

\text{return } \mu

A state is reachable in a program if it appears in some computation of the program starting from a state satisfying the precondition.

ForwardPropagate computes the exact set of reachable states using a (least) fixpoint computation: repeated symbolic execution of program starting from \( \bot \) until equilibrium.
ForwardPropagate(pre, ℒ)

$W := \{ L_{init} \}$

$\mu(L_{init}) := pre$

foreach $L \in ℒ\{L_{init}\}$ do $\mu(L) := \bot$

while $W \neq \phi$ do

pick $L_j \in W$; $W := W\{L_j\}$

foreach $L_k \in succ(L_j)$ do

if $\neg(sp(S_j; ...; S_k, \mu(L_j)) \Rightarrow \mu(L_k))$

then

$\mu(L_k) := \mu(L_k) \lor sp(S_j; ...; S_k, \mu(L_j))$

$W := W \cup \{L_k\}$

return $\mu$

A state is reachable in a program if it appears in some computation of the program starting from a state satisfying the precondition

Inductive assertions usually over-approximate the set of reachable states: every reachable state satisfies the annotation, but other unreachable states can also satisfy the annotation
Abstract interpretation can help force termination!
Abstract interpretation can help force termination!

Ensures implication checking is decidable by manipulating abstract states --- an artificially constrained form of state sets.

 Helps termination of main loop via widening --- guessing a limit overapproximation to a sequence of state sets.

$\bigcap \{\text{true}\} \bigcup \{\text{true}\}$
Type Systems
Deductive Verification
Model Checking
Abstract Interpretation

Interactive theorem provers
Automatic theorem provers

Static analysis

SAT/SMT solvers
Abstraction Interpretation


A framework for designing sound-by-construction static analyses
Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain
Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain.
Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain
Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain.
Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain.
Soundness: If abstract program is safe, then concrete program is safe

No false negative (no false declaration of absence of errors)
If abstract program is unsafe, then concrete program may still be safe. **False positive** (false alarm)
Ideally, we want the most precise (strongest) overapproximation of reachable states expressible in the abstract domain of choice.
Choose abstract domain based on what you want to prove about the program!
Step 1: Abstract domain

An abstract domain is just a set of abstract values we want to track

Define an abstract domain $D$ as a syntactic class of $\Sigma_T$-formulas of some theory $T$.
Each member $\Sigma_T$-formula represents a set of concrete states/elements.

In addition, every abstract domain contains:

- $\top$ (top): “don't know”, represents any value
- $\bot$ (bottom): represents empty-set
Define functions that map sets of concrete states/elements to an element of the abstract domain

Define abstraction function $\alpha: \text{FOL} \mapsto D$ to map a FOL formula $F$ to element $\alpha(F')$ of $D$ such that $F \Rightarrow \alpha(F')$
1. $D_I$: Interval domain

$\top$

$T$

$\bot$

Conjunctions of $\mathbb{Q}$-literals

$u \leq c$, $u \geq c$

\begin{align*}
&\text{Interval arithmetic, } [l, u] \\
&\text{var} \quad \text{const}
\end{align*}

2. $\chi(F)$

\[ F : \begin{cases}
T & \text{otherwise} \\
\frac{1}{a}, b \in \mathbb{N}, a > 0
\end{cases} \]

$\begin{align*}
&u \leq b - F: au \leq b \\
&u \geq \frac{1}{a} - F: au \geq b \\
&w \leq b - F: au = b \\
&v \geq \frac{b}{a} \quad \text{Literals}
\end{align*}
Step 3: Abstract transformer (SP)

Define abstract semantics/transformers for each statement

Define an abstract strongest postcondition $sp_D$ for assumption and assign statements such that

$$sp(S, P) \Rightarrow sp_D(S, P) \text{ and } sp_D(S, P) \in D$$
Step 3: Abstract transformer (SP)

Recall: $sp(\text{assume } C, P) = C \land P$

Define abstract conjunction $\prod_D$ such that for $F_1, F_2 \in D$

$$F_1 \land F_2 \Rightarrow F_1 \prod_D F_2 \text{ and } F_1 \prod_D F_2 \in D$$

Then for $F \in D: sp_D(\text{assume } C, F) = \alpha(C) \prod_D F$

Recall: $sp(x := E, P) = \exists x^0. x = E[x^0/x] \land p[x^0/x]$

The existential quantification can lead to a quantifier alternation during implication checking and is undesirable
3. \( \Pi_I: \text{exact} \)

\[
[x_1, u_1] \Pi_I [x_2, v_2] = \begin{cases} 
[\infty, -\infty] & \text{if } \max(x_1, x_2) > \min(u_1, v_2) \\
[\max(x_1, x_2), \min(u_1, v_2)] & \text{otherwise}
\end{cases}
\]

\[sp_I (\text{assume } C, F) = \chi(C) \Pi_I F \]

\[sp_I (x := a, F) = ?? \]

Interval arithmetic
Step 4: Abstract disjunction

Define abstract disjunction $\sqcup_D$ such that for $F_1, F_2 \in D$

$$F_1 \lor F_2 \Rightarrow F_1 \sqcup_D F_2 \text{ and } F_1 \sqcup_D F_2 \in D$$
Step 5: Abstract implication checking

When selecting $D$ ensure that the implication in the validity check is decidable
4. $W_I : \text{interval hull}$

$[x_1, y_1] \cup_I [x_2, y_2] = [\min (x_1, x_2) \cup \min (y_1, y_2), \max (x_1, x_2) \cup \max (y_1, y_2)]$

5. $f, g_I \in D_I$

Validity of $f \Rightarrow g_I$ is decidable!

$[x_1, y_1] \leq [x_2, y_2]$
Defining an abstraction does not suffice to guarantee termination. Some abstract domains need to be equipped with a widening operator.

Define a widening operator $\nabla_D : D \times D \mapsto D$ such that

1. for all $F_1, F_2 \in D$, $F_1 \lor F_2 \Rightarrow F_1 \nabla_D F_2$ and
2. for all infinite sequences $F_1, F_2, \ldots$ with $F_1 \Rightarrow F_2 \Rightarrow \ldots$, the infinite sequence $G_1 = F_1, \ldots, G_{i+1} = G_i \nabla_D F_{i+1}, \ldots$ converges.

Thus, the sequence $G_i$ converges even if the sequence $F_i$ does not.

**Step 6: Widening**
6. \( F: [x_1, u_1] \rightarrow [x_2, u_2] \)

\[ F \cap G = [x, u] = [x_1, u_1] \cup [x_2, u_2] \]

\[ x = \begin{cases} -\infty, & \text{if } x_2 < x_1 \\ x_1, & \text{otherwise} \end{cases} \]

\[ u = \begin{cases} \infty, & \text{if } u_2 > u_1 \\ u_1, & \text{otherwise} \end{cases} \]

\( F \cap G \) drops "bounds that grow from \( F \) to \( G \)"
**AbstractForwardPropagate**(pre, ℒ)

\[ W := \{ \text{L}_{\text{init}} \} \]

\[ \mu(\text{L}_{\text{init}}) := \alpha(\text{pre}) \]

foreach \( L \in \mathcal{L}\setminus \{\text{L}_{\text{init}}\} \) do \( \mu(L) := \bot \)

while \( W \neq \phi \) do

\[ \text{pick } L_j \in W; \ W := W\setminus \{L_j\} \]

foreach \( L_k \in \text{succ}(L_j) \) do

\[ \text{if } \neg(\text{sp}_D(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \]

\[ \text{then if WIDEN()} \]

\[ \text{then } \mu(L_k) := \mu(L_k) \sqcap_D (\mu(L_k) \sqcup_D \text{sp}_D(S_j; \ldots; S_k, \mu(L_j))) \]

\[ \text{else } \mu(L_k) := \mu(L_k) \sqcup_D \text{sp}(S_j; \ldots; S_k, \mu(L_j)) \]

\[ W := W \cup \{L_k\} \]

return \( \mu \)
AbstractForwardPropagate(\(\text{pre}, \mathcal{L}\))

\[ W := \{L_{\text{init}}\} \]

\[ \mu(L_{\text{init}}) := \alpha(\text{pre}) \]

foreach \(L \in \mathcal{L} \setminus \{L_{\text{init}}\}\) do \(\mu(L) := \bot\)

while \(W \neq \emptyset\) do

pick \(L_j \in W\); \(W := W \setminus \{L_j\}\)

foreach \(L_k \in \text{succ}(L_j)\) do

if \(\neg \text{sp}_D(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)) \)

then if \(\text{WIDEN()}\)

then \(\mu(L_k) := \mu(L_k) \lor \text{sp}_D (S_j; \ldots; S_k, \mu(L_j)))\)

else \(\mu(L_k) := \mu(L_k) \lor \text{sp} (S_j; \ldots; S_k, \mu(L_j)))\)

\(W := W \cup \{L_k\}\)

return \(\mu\)
AbstractForwardPropagate(pre, $\mathcal{L}$)

$W := \{L_{init}\}$

$\mu(L_{init}) := \alpha(\text{pre})$

foreach $L \in \mathcal{L} \setminus \{L_{init}\}$ do $\mu(L) := \bot$

while $W \neq \emptyset$ do

pick $L_j \in W$; $W := W \setminus \{L_j\}$

foreach $L_k \in \text{succ}(L_j)$ do

if $\neg(s_p_D(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k))$

then if WIDEN()

then $\mu(L_k) := \mu(L_k) \lor_D (\mu(L_k) \lor_D s_p_D(S_j; \ldots; S_k, \mu(L_j)))$

else $\mu(L_k) := \mu(L_k) \lor_D s_p(S_j; \ldots; S_k, \mu(L_j))$

$W := W \cup \{L_k\}$

return $\mu$
AbstractForwardPropagate\((pre, \mathcal{L})\)

\[
W := \{L_{init}\}
\]

\[
\mu(L_{init}) := \alpha(pre)
\]

foreach \(L \in \mathcal{L}\setminus\{L_{init}\}\) do
\[
\mu(L) := \bot
\]

while \(W \neq \emptyset\) do

pick \(L_j \in W\); \(W := W\setminus\{L_j\}\)

foreach \(L_k \in \text{succ}(L_j)\) do

if \(\neg (sp_D(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k))\)
then

if \(\text{WIDEN()}\)
then
\[
\mu(L_k) := \mu(L_k) \triangledown_D (\mu(L_k) \sqcup_D sp_D(S_j; \ldots; S_k, \mu(L_j)))
\]
else
\[
\mu(L_k) := \mu(L_k) \sqcup_D sp(S_j; \ldots; S_k, \mu(L_j))
\]

\(W := W \cup \{L_k\}\)

return \(\mu\)
Invariant generation

- Today: least fixed-point in an abstract domain
- Farkas’ lemma (for linear invariants) [Colon_CAV_2003]
- Interpolation [McMillan_TACAS_2008]
- Abductive inference [Dillig_OOPSLA_2013]
- IC3 [Bradley_VMCAI_2011]
- Machine learning [Garg_CAV_2014]
Abstraction Interpretation


A framework for designing sound-by-construction static analyses
Lattices and Abstract Domains

Abstraction function $\alpha: \text{FOL} \mapsto D$ maps a FOL formula $F$ to element $\alpha(F)$ of $D$ such that $F \Rightarrow \alpha(F)$

Concretization function $\gamma: D \mapsto \text{FOL}$ to map an element of $D$ to a FOL formula $F$
Lattices and Abstract Domains

Abstraction function $\alpha: \text{FOL} \mapsto D$ maps a FOL formula $F$ to element $\alpha(F)$ of $D$ such that $F \Rightarrow \alpha(F)$

Concretization function $\gamma: D \mapsto \text{FOL}$ to map an element of $D$ to a FOL formula $F$

- Concretization function defines partial order $\subseteq$ on abstract values:
  $A_1 \subseteq A_2$ iff $\gamma(A_1) \Rightarrow \gamma(A_2)$
- In an abstract domain $D$, every pair of elements has a lub ($\sqcup_D$) and glb ($\sqcap_D$)

$\Rightarrow$ Mathematical Lattice
Galois insertion

Important property of the abstraction and concretionization function is that they are almost inverses:

$$\alpha(\gamma(A)) = A$$

$$C \subseteq \gamma(\alpha(C))$$

Captures soundness of the abstraction
\begin{enumerate}
\item \texttt{x := 0} \\
\item \texttt{y := 1} \\
\item \textbf{while} \{I\} (y \leq n) \textbf{do} \\
\hspace{1em} \textbf{if} (z == 0) \textbf{then} \\
\hspace{2em} \texttt{x := x + 1} \\
\hspace{1em} \textbf{else} \\
\hspace{2em} \texttt{x := x + y} \\
\hspace{1em} \texttt{y := y + 1} \\
\item \textbf{end while} \\
\end{enumerate}
Let’s do a static analysis of this program under the “sign” abstract domain.
Sign domain

1. D: T

\[ P - \text{pos} = \left\{ x : x \in \mathbb{Z} \land x > 0 \right\} \]

\[ N - \text{neg} = \left\{ x : x \in \mathbb{Z} \land x < 0 \right\} \]

\[ Z - \text{zero} = \left\{ 0 \right\} \]

\[ NN - \text{non-neg} = \left\{ x : x \in \mathbb{Z} \land x \geq 0 \right\} \]

2. \( \alpha \left( \left\{ 2, 5, 0 \right\} \right) = NN \quad \alpha \left( \left\{ \frac{3}{2}, 5 \right\} \right) = P \]

\( \alpha \left( \left\{ -2, 2 \right\} \right) = I \)
Let’s do a static analysis of this program under the “sign” abstract domain.

Here \textbf{pre} and \textbf{post} are not given.

Assume \textbf{pre} is \textbf{true}. This analysis will find an overapproximation of reachable states at all cut-points.

One can then easily verify if some given \textbf{post} holds.
\{x = Z, y = P\} \rightarrow \{x = Z, y = P\} \rightarrow \{x = \bot, y = \bot\} \rightarrow \{x = Z, y = P\} \rightarrow \{x = \bot, y = \bot\} \rightarrow \{x = \bot, y = \bot\}

\begin{align*}
&x := 0 \\
y := 1
\end{align*}
\{x = Z, y = P\} \quad x := 0 \quad y := 1

\{x = Z, y = P\} \quad \text{loop head}

\{x = Z, y = P\} \quad y > n \quad y \leq n

\{x = Z, y = P\} \quad \text{exit}

\{x = Z, y = P\} \quad \text{conditional}

\{x = Z, y = P\} \quad \text{loop end}
\[ x := 0 \\
\] \[ y := 1 \]

\{x = Z, y = P\}

\text{loop head}

\text{conditional}

\{x = Z, y = P\}

z \neq 0 \quad z = 0

\{x = P, y = P\}

\text{x := x + y}
\text{x := x + 1}

\{x = P, y = P\}

\text{loop end}

\text{exit}
\begin{align*}
\{ x = Z, y = P \} \quad \{ x = Z, y = P \} \quad \{ x = Z, y = P \} \\
\{ x = Z, y = P \} \quad \text{loop head} \quad \text{loop end} \\
\{ x = P, y = P \} \quad \{ x = P, y = P \} \quad \{ x = P, y = P \}
\end{align*}
\{x = \mathbb{Z}, y = P\} \\
\{x = \mathbb{N}, y = P\} \\
\{x = \mathbb{Z}, y = P\} \\
{\{x = \mathbb{P}, y = P\}}
$x := 0$
$y := 1$

loop head

{$x = Z, y = P$}

{$x = NN, y = P$}

exit

conditional

{$x = NN, y = P$}

$y > n$

$y \leq n$

$z \neq 0$

$z = 0$

$x := x + y$

$x := x + 1$

{$x = P, y = P$}

{$x = P, y = P$}

{$x = P, y = P$}

loop end

{$x = P, y = P$}
\[
\begin{align*}
\text{loop head} & : \quad x := 0 \\
& \quad y := 1
\end{align*}
\]

\[
\begin{align*}
\{x = Z, y = P\} & \quad y > n \\
\{x = NN, y = P\} & \quad y \leq n \\
\{x = NN, y = P\} & \quad z \neq 0 \\
\{x = P, y = P\} & \quad z = 0
\end{align*}
\]

\[
\begin{align*}
x & := x + y \\
& \quad x := x + 1 \\
& \quad y := y + 1 \\
\text{loop end}
\end{align*}
\]
\{x = Z, y = P\}

\{x = NN, y = P\}

\{x = NN, y = P\}

\{x = P, y = P\}

Fixpoint!
Partial specifications

- Pre/postcondition pairs are also called **functional specifications**
  - Meant to be **complete**
- Writing complete specifications can be very hard
- Partial specifications are easier to write and, often, easier to check
- Partial specifications may be explicit or implicit
Partial specifications

- Pre/postcondition pairs are also called **functional specifications**
  - Meant to be **complete**
- Writing complete specifications can be very hard
- **Partial specifications** are easier to write and, often, easier to check
- Partial specifications may be explicit or implicit

- **assertions**
- no division-by-zero
- no null-pointer-dereference
An **assertion** is a FOL formula $F$ at a program location $L$. It asserts that $F$ is true whenever program control reaches $L$. Equivalently, it asserts that the set of reachable states at $L$ satisfies $F$. 

```
x := 0
y := 1
while (y <= n) do
  if (z == 0) then
    x := x + 1
  else
    x := x + y
  assert y > 0
y := y + 1
```
An assertion is a FOL formula $F$ at a program location $L$. It asserts that $F$ is true whenever program control reaches $L$. Equivalently, it asserts that the set of reachable states at $L$ satisfies $F$.

Can use forward propagation to also compute the exact/overapproximate set of reachable states at location $L$ (even if $L$ is not a cutpoint).
\[ x := 0 \]
\[ y := 1 \]
while \((y \leq n)\) do
  if \((z = 0)\) then
    \[ x := x + 1 \]
  else
    \[ x := x + y \]
    assert \(y > 0\)
  end
\[ y := y + 1 \]
end

```plaintext
\{x = \bot, y = \bot\}
\{x = \bot, y = \bot\}
\{x = \bot, y = \bot\}
```
\[
x := 0 \\
y := 1 \\
\text{while } (y \leq n) \text{ do} \\
\quad \text{if } (z = 0) \text{ then} \\
\quad \quad x := x + 1 \\
\quad \text{else} \\
\quad \quad x := x + y \\
\quad \text{assert } y > 0 \\
\quad y := y + 1 \\
\text{end while}
\]
\(x := 0\)
\(y := 1\)

while (\(y <= n\)) do
  if (\(z == 0\)) then
    \(x := x + 1\)
  else
    \(x := x + y\)
  assert \(y > 0\)
  \(y := y + 1\)

loop head

exit

conditional

loop end
\[ \begin{align*}
    &x := 0 \\
    &y := 1 \\
    \text{loop head} \\
    &y > n \quad \text{exit} \\
    &y \leq n \quad \text{conditional} \\
    \{x = Z, y = P\} \\
    &x := x + y \\
    &x := x + 1 \\
    \{x = P, y = P\} \\
    &y := y + 1 \quad \text{loop end} \\
    \{x = P, y = P\} \\
    &y = P \\
    \{x = P, y = P\} \\
    \{x = P, y = P\} \\
    \{x = P, y = P\} \\
    \{x = Z, y = P\} \\
    \{x = Z, y = P\} \\
\end{align*} \]
\{x = Z, y = P\}

\{x = Z, y = P\}

\{x = Z, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}
\{x = Z, y = P\}

\{x = NN, y = P\}

\{x = NN, y = P\}

Fixpoint!

\{y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}

\{x = P, y = P\}
\[
\begin{align*}
&\{x = Z, y = P\} \\
&\{x = NN, y = P\} \\
&\{x = NN, y = P\} \\
&\{y = P\} \\
&\{x = P, y = P\} \\
&\{x = P, y = P\} \\
&\text{Abstract program is correct}
\end{align*}
\]
\[
\begin{aligned}
    x &:= 0 \\
y &:= 1 \\
    \text{while } (y \leq n) \text{ do} \\
        \quad \text{if } (z = 0) \text{ then} \\
        \qquad x := x + 1 \\
        \quad \text{else} \\
        \qquad x := x + y \\
        \quad \text{assert } y > 0 \\
    \end{aligned}
\]
x := 0
y := 1
while (y <= n) do
  if (z == 0) then
    x := x + 1
  else
    x := x + y
  y := y + 1

Always y > 0
A safety property $F$ asserts that $F$ is true at every program location in every program execution. Equivalently, it asserts that the set of reachable states at any location satisfies $F$.

Always $y > 0$
A safety property $F$ asserts that $F$ is true at every program location in every program execution. Equivalently, it asserts that the set of reachable states at any location satisfies $F$.

Can use forward propagation to compute the exact/overapproximate set of reachable states at every location $L$.

Always $y > 0$
Always $y = P$
\begin{verbatim}
    x := 0
    y := 0
    while (y <= n) do
        if (z == 0) then
            x := x + 1
        else
            x := x + y
        y := y + 1
    end while
\end{verbatim}

Always \( y > 0 \)
Summary

Today

- Automated invariant generation
- Forward propagation using strongest postconditions
- Abstract interpretation

Next

- Abstraction-refinement and predicate abstraction