Program invariance proofs

Class 8, Friday 2015-10-23, 5:10–7:00 PM, WWH–312

Patrick Cousot
pcousot@cs.nyu.edu  cs.nyu.edu/~pcousot

Formal reasoning on programs

• “Reasoning on a program” means reasoning on all its executions in all possible execution environments/contexts
• “Formal” means mathematical
• So we need to mathematically define:
  • A model of program executions (called the semantics)
  • A notion of program property (invariance for our examples)
  • A proof method to show that a program semantics satisfies a given (invariance) property

Introduction

A (classical) invariance proof example
Euclid’s greatest common divisor algorithm

\{ a = a_0, b = b_0 \in \mathbb{Z} \land a_0 > 0 \land b_0 > 0 \}  \quad \text{(hypothesis)}

\{ \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \}  \quad \text{(by hypothesis)}

\textbf{while} a \neq b \textbf{ do}

\{ \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \land (a > b > 0 \lor 0 < a < b) \}  

\textbf{if} a > b

\{ \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \land (a > b > 0) \}  

a := a - b  

\{ \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \land (a > 0 \land b > 0) \}  \quad \text{(since GCD}(a,b) = \text{GCD}(a-b,b) \text{ when } a > b)  

\text{else}

\{ \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \land (b > a > 0) \}  

b := b - a  

\{ \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \land (b > 0 \land a > 0) \}  \quad \text{(since GCD}(a,b) = \text{GCD}(a,b-a) \text{ when } b > a)  

\textbf{od};

\{ a = b = \text{GCD}(a,b) = \text{GCD}(a_0,b_0) \}  \quad \text{(since GCD}(a,a) = a)  

\textbf{return } a

How can we justify this (Turing/Naur/Floyd/Hoare) proof method?

Transition semantics (reminder)

- The transition relation is 
  \[ \tau \in P(\delta \times \mathcal{A} \times \delta') \]

- \(<s,a,s'> \in \tau\) if and only if an execution of action \(a\) in state \(s\) may yield next state \(s'\)

- In general a state \(s\) has many possible successor states \(s'\) for a given action \(a\) (non-determinism) e.g. \(x := ?;\)

- We write \(s \xrightarrow{a} s'\) for \(<s,a,s'> \in \tau\)

States

- \(\mathcal{V}\): set of variables

- \(\mathcal{V}'\): set of values of variables (e.g. \(\mathbb{Z} \subseteq \mathcal{V}'\))

- \(\mathcal{E} := \mathcal{V} \rightarrow \mathcal{V}'\), environments/memory states

- \(\mathcal{A}\), actions

- \(\mathcal{C}\), commands

- \(\mathcal{G} := \mathcal{C} \cup \{\varepsilon\}\), control states

- \(\mathcal{S} := \mathcal{G} \times \mathcal{E}\), states
Title

• \(<\text{skip}, \rho> = \text{skip} \Rightarrow \epsilon, \rho>\)

Assignment

• \(\nu \in E[E][\rho]\)

\(<x := E, \rho> = x := E \Rightarrow \epsilon, \rho[x := \nu]>\)

• \(\rho[x := \nu](y) := \rho(y)\) when \(y \neq x\)

\(\rho[x := \nu](x) := \nu\)

\(\rho[x := \nu](y) := \rho(y)\) when \(y \neq x\)

\(\rho[x := \nu](x) := \nu\)

Well-defined by structural induction

Examples:

\(<x := x + 1, \rho> \Rightarrow x := x + 1 \Rightarrow \epsilon, \rho[x := \rho(x) + 1]>\)

\(<x := x + 2, \rho[x := \rho(x) + 1]>\)

\(\rho[x := \rho(x) + 3]>\)

• \(\text{true} \in B[B][\rho]\)

\(<\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}, \rho> = B \Rightarrow C_1, \rho>\)

• \(\text{false} \in B[B][\rho]\)

\(<\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}, \rho> = \neg B \Rightarrow C_2, \rho>\)

Well-defined by structural induction

• Non-deterministic if \(B\) is non-deterministic
Trace semantics (reminder)

Traces

- A trace $t \in \mathcal{T}$ is a non-empty finite or infinite sequence of states $s_i$ separated by actions $a_i$

  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots s_n \quad (n \geq 0)$

  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots s_n \xrightarrow{a_n} s_{n+1} \ldots$

- The states describe the control, values of variables, past outputs and future effective inputs

- All actions effectively contribute to computations

Example

- int x;
  input(x);
  x := x+1;

- < int x; input(x); x := x+1; | input:5; output: >
  - int x ->
  < input(x); x := x+1; | x:1789; input:5; output: >
  - input(x) ->
  < x := x+1; | x:5; input::output: >
  - x := x+1 ->
  <ε | x:6; input::output: >
Trace semantics abstraction

1. The trace semantics of a program is a rather precise and detailed description of its possible executions.

2. We are often interested in less precise descriptions of these executions, that is in abstractions of the trace semantics.

3. To illustrate the concept of trace semantics abstraction, we now consider an example where we are only interested in knowing in which final states program executions do terminate, for a given initial state (thus ignoring all intermediate steps of the finite computations as well as non-terminating executions).

Example of trace abstraction

- We may be interested in pre-post properties which consist in observing the initial and final states of finite traces only and excluding control:

  \[<1;\text{input:}5;\text{output:}> \quad \text{int} \quad x \rightarrow <2;|x:1789;\text{input:}5;\text{output:}> - \text{input}(x) \rightarrow <3;|x:5;\text{input:};\text{output:}> - x := x+1 \rightarrow <4;|x:6;\text{input:};\text{output:}> \quad \alpha \rightarrow \begin{cases} <\text{input:}5>,<x:6;\text{output:}> \end{cases}\]

- All intermediate states and actions in the computation as well as control, initial empty output, and final (empty) input are disregarded in the pre-post abstraction.

Example of trace abstraction (cont’d)

- The pre-post trace abstraction is

  \[\alpha (<c_0,m_0> - a_0 \rightarrow s_1 - a_1 \rightarrow s_2 \ldots <c_n,m_n>) := <m_0,m_n>\]
Trace abstraction

- A trace semantics describes possible program executions at a rather precise level of observation.
- In general, we are not interested in all such details and would like to discuss program executions at more abstract levels of observation.
- These trace abstractions $\alpha$ can be defined formally.

Semantic abstraction

- A program semantics $S$ is a set of traces so $S \subseteq T$ i.e. $S \in \wp(T)$ (where $\wp(T) := \{X \mid X \subseteq T\}$ is the powerset of the set $T$).
- The trace abstraction $\alpha \in T \longmapsto \wp$ induces a semantic abstraction $\alpha \in \wp(T) \longmapsto \wp(\wp$) defined as:

$$\alpha(S) := \{\alpha(t) \mid t \in S\}$$

Example of semantic abstraction (cont’d)

- The pre-post semantics abstraction is

$$\tilde{\alpha}(S) := \{\tilde{\alpha}(s_0 - a_0 \rightarrow \ldots s_n) \mid s_0 - a_0 \rightarrow \ldots s_n \in S\}$$

(so that infinite traces for non-terminating executions are eliminated)
Observation concretization

• The adjoint semantic concretization \( \gamma \in \wp(\Theta) \leftarrow \wp(\mathcal{T}) \) provides the \( \subseteq \)-largest semantics with given observation \( O \in \wp(\Theta) \)

\[ \gamma(O) := \{ t \in \mathcal{T} \mid \alpha(t) \in O \} \]

• \( \gamma(O) \) is the \( \subseteq \)-largest semantics with given observation \( O \) in that (see later):
  - \( \alpha(\gamma(O)) = O \)
  - if \( \alpha(S) = O \) then \( S \subseteq \gamma(O) \)

Galois connection/insertion

• The semantic abstraction \( \langle \alpha, \gamma \rangle \) is a Galois connection, that is, by definition

\[ \forall S \in \wp(\mathcal{T}): \forall O \in \wp(\Theta): \alpha(S) \subseteq O \iff S \subseteq \gamma(O) \]

• Moreover, the semantic abstraction \( \langle \alpha, \gamma \rangle \) is a Galois retraction/insertion, that is, by definition

\[ \alpha \text{ is surjective} \]

Program semantic properties

Mathematical properties

• Mathematical properties can be defined
  - informally, more or less rigorously in english (to be an even number means divisible by 2),
  - logically, by a logical formula (even(n) := \( \exists k : n = 2k \))
  - set-theoretically, as the set of elements that have the property \( \{..., -4, -2, 0, 2, 4, ... \} \)
  - We use all forms, mainly the set-theoretical definitions
Program property, informally

- by program property, we mean a property of its semantics
- So a program property can be defined as a set of semantics that is a set of sets of traces
- We can also consider properties of abstractions of semantics that is a set of abstractions of sets of traces

Example

- The property of programs that always terminate with the same result which can be either 0 or 1

Examples: `print(1);` or `int x; x:=0; print(x);`

Counter-examples: `while true do skip;` or `bool b; read(b); if b print(0) else print 1;`

Program property, formally

- We let $T$ be the set of execution traces (finite or infinite sequences of states in $S$ separated by actions in $A$)
- A semantics $S$ is a set of traces so $S \in \wp(T)$
- A semantic property $P$ is a set of semantics so $P \in \wp(\wp(T))$

Satisfaction

- By definition, a semantics $S \in \wp(T)$ satisfies a semantic property $P \in \wp(\wp(T))$, if and only if $S \in P$
- Example: the semantics

satisfies
Program trace properties

Example

- P =

- $\alpha_t(P) =$

Trace property abstraction

- The trace property abstraction $\alpha_t$ abstracts a semantic property $P \in \wp(\wp(\mathcal{T}))$ into a trace property $\alpha_t(P) \in \wp(\mathcal{T})$.

- The trace property abstraction is defined as

$$\alpha_t \in \wp(\wp(\mathcal{T})) \mapsto \wp(\mathcal{T})$$

$$\alpha_t(P) := \bigcup P := \bigcup \{ S \in \wp(\mathcal{T}) \mid S \in P \}$$

Example (cont’d)

- The programs

  print(1);
  int x; x:=0; print(x);

  satisfy the semantic property $P$ while the programs

  while true do skip; or
  bool b; read(b); if b print(0) else print 1;

  do not satisfy this semantic property $P$.

- However, all of these programs do satisfy the trace property $\alpha_t(P)$

- Semantic properties are more precise than traces properties
Galois connection

- \( <\alpha_t, \gamma_t> \) is a Galois connection, where

\[
\gamma_t(Q) := \{ S \in \mathcal{G}(T) \mid S \subseteq Q \}
\]

Proof

For all \( P \in \mathcal{G}(\mathcal{G}(T)) \) and \( Q \in \mathcal{G}(T) \),

\[
\alpha_t(P) \subseteq Q \\
\Leftrightarrow \bigcup P \subseteq Q \quad \text{def. } \alpha_t \\
\Leftrightarrow \bigcup \{ S \mid S \in P \} \subseteq Q \quad \text{def. } \bigcup \\
\Leftrightarrow \forall S \in P: S \subseteq Q \quad \text{def. } \subseteq \\
\Leftrightarrow P \subseteq \{ S \mid S \subseteq Q \} \quad \text{def. } \forall \\
\Leftrightarrow P \subseteq \gamma_t(Q) \quad \text{def. } \gamma_t
\]

Satisfaction

- By definition, a trace semantics \( S \in \mathcal{G}(T) \) is said to “satisfy an abstract property \( Q \)” if and only if it satisfies the concretization \( \gamma_t(Q) \in \mathcal{G}(\mathcal{G}(T)) \) of this abstract property \( Q \), that is \( S \in \gamma_t(Q) \). Equivalently,

- A semantics \( S \in \mathcal{G}(T) \) satisfies an abstract trace property \( Q \in \mathcal{G}(T) \) if and only if \( S \subseteq Q \).

Proof

\[
S \in \gamma_t(Q) \\
\Leftrightarrow S \in \{ S' \mid S' \subseteq Q \} \quad \text{def. } \gamma_t \\
\Leftrightarrow S \subseteq Q \quad \text{def. } \subseteq
\]

Example

- The semantics

\[
\text{satisfies the trace property}
\]
Note
• $\subseteq$ exactly corresponds to logical implication $\implies$
Example: reachability

• Trace property

\[ Q = s_0 \rightarrow \alpha_0 \rightarrow s_1 \rightarrow \alpha_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \]

• Reachability abstraction:

\[ \alpha_r(Q) = s_0 \quad s_1 \quad s_2 \quad \ldots \quad s_n \]

\[ s_0' \quad s_1' \quad s_2' \quad \ldots \quad s_n' \quad s_{n+1}' \quad \ldots \]

The actions and the order of appearance of states during computations are lost by the abstraction.

Galois connection

• \( <\alpha_r, \gamma_r> \) is a Galois connection, where

\[ \gamma_r(I) := \{ s_0 \rightarrow \alpha_0 \rightarrow s_1 \rightarrow \alpha_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \in \mathcal{T} |\]

\[ \forall i \in [0, n\ldots]: s_i \in I \} \]

\(^{(*)}\) The notation \([0,n\ldots]\) is meant to denote \([0,n]\) for finite traces with last state \(s_n\) and \([0, +\infty), +\infty\) excluded, for infinite traces

Initial states abstraction

• In general, we are interested in executions starting from states satisfying an initial condition \(\mathcal{I} \in \mathcal{G}(\mathcal{S})\)

• The initial states abstraction is defined as

\[ \alpha_{\mathcal{S}}(Q) := \{ s_0 \rightarrow \alpha_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \in Q | s_0 \in \mathcal{I} \} \]

• This is a Galois connection

Proof

• Forall \(Q \in \mathcal{G}(\mathcal{T})\) and \(I \in \mathcal{G}(\mathcal{S})\), we have

\[ \alpha_r(Q) \subseteq I \]

\[ \Leftrightarrow \{ s \in \mathcal{S} | \exists s_0 \rightarrow \alpha_0 \rightarrow \ldots \rightarrow s_n \in Q: \exists i: s = s_i \} \subseteq I \] \(\text{def.} \alpha_r\)

\[ \Leftrightarrow \forall s \in \mathcal{S}: \forall s_0 \rightarrow \alpha_0 \rightarrow \ldots \rightarrow s_n \in Q: \forall i: (s = s_i) \Rightarrow (s \in I) \] \(\text{def.} \subseteq\)

\[ \Leftrightarrow \forall s_0 \rightarrow \alpha_0 \rightarrow \ldots \rightarrow s_n \in Q: \forall i: s_i \in I \] \(\text{def.} \subseteq\)

\[ \Leftrightarrow Q \subseteq \{ s_0 \rightarrow \alpha_0 \rightarrow \ldots \rightarrow s_n \ | \forall i \in [0, n\ldots]: s_i \in I \} \] \(\text{def.} \subseteq\)

\[ \Leftrightarrow Q \subseteq \gamma_r(I) \] \(\text{def.} \gamma_r\)
Example: initial states

• Trace property

\[ Q = \{ s_0 \rightarrow a_0 \rightarrow s_1 \rightarrow a_1 \rightarrow s_2 \ldots s_n, s'_0 \rightarrow a'_0 \rightarrow s'_1 \rightarrow a'_1 \rightarrow s'_2 \ldots s'_m \} \]

• Initial states abstraction:

\[ \alpha_F(Q) = \{ s_0 \rightarrow a_0 \rightarrow s_1 \rightarrow a_1 \rightarrow s_2 \ldots s_n, s'_0 \rightarrow a'_0 \rightarrow s'_1 \rightarrow a'_1 \rightarrow s'_2 \ldots s'_m \} \]

Reachability from initial states

• The reachability abstraction from initial states \( F \in \mathcal{G}(F) \) is

\[ \alpha_r \circ \alpha_F \]

(where \( f \circ g (x) := f(g(x)) \) is the composition of functions)

• This is a Galois connection (since the composition of Galois connections is a Galois connection)

Satisfaction

• A trace semantics \( S \in \mathcal{G}(F) \) is said to "satisfy a state property" \( I \in \mathcal{G}(F) \) if and only if

\[ S \in \gamma_r( \gamma_r(I) ) \]

\[ S \subseteq \gamma_r(I) \]

\[ \alpha_r(S) \subseteq I \]

that is if the reachable states of \( S \) are all in \( I \).

• \( I \) is called an "invariant" of \( S \) (i.e. program \( P \))

Example of invariant

The invariant \( I \) include all reachable states from the initial states \( F \) plus some unreachable states such as \( s''_1, s''_2, \ldots, \) and \( s''_i \).
Example of program invariant

1: x := 0;
2: while (x <= 10) {
   3: x := x + 1;
4: }
5:

Invariant:

\[ I = \{<1:x_i> | i \in \mathbb{Z}\} \cup \{<2:x_i> | 0 \leq i \leq 11\} \cup \{<3:x_i> | 0 \leq i \leq 10\} \cup \{<4:x_i> | 1 \leq i \leq 11\} \cup \{<5:x_{11}>\} \]

In logical form (\(e\) is the program counter):

\[ I(e, x) = (e=1 \land x \in \mathbb{Z}) \lor (e=2 \land 0 \leq x \leq 11) \lor (e=3 \land 0 \leq x \leq 10) \lor (e=4 \land 1 \leq x \leq 11) \lor (e=5 \land x = 11) \]

Local invariants

- A global invariant \(I \in \mathcal{P}(\mathcal{E} \times \mathcal{M})\) can be isomorphically represented as \(I \in \mathcal{E} \rightarrow \mathcal{P}(\mathcal{M})\) by attaching local invariants to program points

Example:

\[ I(e, x) := (e=1 \land x \in \mathbb{Z}) \lor (e=2 \land 0 \leq x \leq 11) \lor (e=3 \land 0 \leq x \leq 10) \lor (e=4 \land 1 \leq x \leq 11) \lor (e=5 \land x = 11) \]

is isomorphically represented as

\[ I_1(x) := x \in \mathbb{Z} \]
\[ I_2(x) := 0 \leq x \leq 11 \]
\[ I_3(x) := 0 \leq x \leq 10 \]
\[ I_4(x) := 1 \leq x \leq 11 \]
\[ I_5(x) := x = 11 \]

Local invariants (cont’d)

- The local invariants can be attached as comments in the program

Example:

\[
/* x \in \mathbb{Z} */
x := 0;
/* 0 \leq x \leq 11 */
while (x <= 10) {
   /* 0 \leq x \leq 10 */
   Loop invariant
   x := x + 1;
   /* 1 \leq x \leq 11 */
}
/* x = 11 */

Turing/Floyd/Naur/Hoare invariance proof method
Invariance proof

- An invariance proof of a state property \( I \in \wp(\mathcal{S}) \), for a program \( P \) with trace semantics \( S \in \wp(\mathcal{T}) \) consist in proving that all reachable states of \( S \) satisfy \( I \) that is \( \alpha_r(S) \subseteq I \).

Turing/Floyd/Naur/Hoare proof method

- **Basis condition:**
  Prove that the local invariant \( I_0 \) at the program entry point does hold (i.e. \( \mathcal{S} \subseteq I_0 \) where \( \mathcal{S} \) is the precondition hypothesis on the initial states)

- **Induction condition:** for each program point \( \epsilon \in \mathcal{E} \):
  - Assume that the local invariant \( I_\epsilon \) does hold
  - Prove that after any possible single computation step from program point \( \epsilon \) to the next program point \( \epsilon' \), the local invariant \( I_{\epsilon'} \) does hold

Example of verification conditions

- 1: /* \( x \in \mathbb{Z} \) */
  \( x := 0; \)
- 2: /* \( 0 \leq x \leq 11 \) */
  while (\( x <= 10 \)) {
- 3: /* \( 0 \leq x \leq 10 \) */
  \( x := x + 1; \)
- 4: /* \( 1 \leq x \leq 11 \) */
- 5: /* \( x = 11 \) */

- at 1: (\( x = 0 \)) \( \implies (0 \leq x \leq 11) \)
- at 2 (true): (\( 0 \leq x \leq 11 \land x <= 10 \)) \( \implies (0 \leq x \leq 10) \)
- at 2 (false): (\( 0 \leq x \leq 11 \land x > 10 \)) \( \implies (x = 11) \)
- at 3: (\( 0 \leq x \leq 10 \land x' = x + 1 \)) \( \implies (1 \leq x' \leq 11) \)
- at 4: (\( 1 \leq x \leq 11 \)) \( \implies (0 \leq x \leq 11) \)
- at 5: no successor.

System of inequations

- 1: /* \( I_1(x) := x \in \mathbb{Z} \) */
  \( x := 0; \)
- 2: /* \( I_2(x) := 0 \leq x \leq 11 \) */
  while (\( x <= 10 \)) {
- 3: /* \( I_3(x) := 0 \leq x \leq 10 \) */
  \( x := x + 1; \)
- 4: /* \( I_4(x) := 1 \leq x \leq 11 \) */
- 5: /* \( I_5(x) := x = 11 \) */

- at 1: (\( x = 0 \)) \( \implies I_2(x) \)
- at 2 (true): (\( I_2(x) \land x <= 10 \)) \( \implies I_3(x) \)
- at 2 (false): (\( I_2(x) \land x > 10 \)) \( \implies I_5(x) \)
- at 3: (\( I_3(x) \land x' = x + 1 \)) \( \implies I_4(x') \)
- at 4: \( I_4(x) \implies I_2(x) \)
- at 5: no successor.

The local invariant at program point \( k \) is named \( I_k \). The relations between local invariants yield the system of inequations below. The reachability states provide the strongest solution, for which equality holds.

This was the system of in/equations (where the local invariants are unknown) that we were solving with intervals.
Strengthening the invariant

- The following invariant \( I \) is valid but not strong enough to make the proof:

1: /* \( x \in \mathbb{Z} \) */
   \( x := 0; \)
2: /* \( 0 \leq x \leq 1234 \) */
   while \((x <= 10)\) {
3: /* \( 0 \leq x \leq 123 \) */
   \( x := x + 1; \)
4: /* \( 1 \leq x \leq 12 \) */
}  
5: /* \( x = 11 \) */

at 3: \((0 \leq x \leq 123 \land x' = x + 1) \implies (1 \leq x' \leq 12)\)

- If \( I \) is not strong enough (i.e. does not satisfies the invariant inequalities), prove a stronger one \( I' \subseteq I \) to be invariant

Proof of soundness of the invariance proof method

- Let \( s \) be any reachable state of the semantics \( S \)
- Let \( s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots s_n \ldots \) be any trace of \( S \) by which \( s \) can be reached (at \( s_i = s \))
- The basis verification condition states that \( s_0 \in I \)
- Assume that \( s_k \in I \). The inductive verification condition states that by execution of action \( a_k \), the next state \( s_{k+1} \in I \) satisfies the invariant \( I \)
- By induction on \( k \), all states satisfy the invariant \( s_0, s_1, s_2, \ldots, s_n, \ldots \in I \)
- \( s \) is one of them (\( s_i = s \)) so \( s \in I \), Q.E.D.

Soundness of the invariance proof method

- The soundness of the proof methods means that the invariance proof method is correct

- If a state predicate \( I \in \wp(\mathcal{S}) \) satisfies the basis and induction conditions then \( I \) is invariant for \( S \).

Completeness of the invariance proof method

- The completeness of the invariance proof method means that is is always application

- If \( I \in \wp(\mathcal{S}) \) is invariant for \( S \) then there exists a stronger \( I' \subseteq I \) which does satisfies the basis and induction verification conditions.
Completeness of the invariance proof method

- If \( I \) is invariant for the trace semantics \( S \), then, by definition, we have \( \alpha_r(S) \subseteq I \) where \( \alpha_r(S) \) is the set of reachable states of \( S \).
- We have to find \( I' \subseteq I \) such that the basis and induction step are applicable to \( I' \).
- We can always choose \( I' = \alpha_r(S) \) since
  - Basis: the reachable states include the initial states
  - Induction: the successor states of the reachable states by a computation step are reachable.

Floyd/Naur/Hoare verification conditions

- The basis and induction verification conditions can be specified by induction on the syntactic structure of programs (Hoare logic), /*P*/ C /*Q*/ written \{P\}C\{Q\}
  - \( \frac{P \Rightarrow Q}{\{P\} \text{skip} \{Q\}} \) [skip]
  - \( \frac{\exists x': P(x', y) \land x = f(x', y) \Rightarrow Q(x, y)}{\{P(y)\} \ x := f(x, y) \{Q(x, y)\}} \) [:=]
  - \( \frac{\exists x': P(x', y) \Rightarrow \forall x: Q(x, y)}{\{P(x, y)\} \text{input}(x) \{Q(x, y)\}} \) [input]

\( f \) is assumed to have no side-effect (does not modify any variable).

On the relative completeness of Hoare logic

- We have used sets to describe invariants
- One could also require the invariants to be expressed in a given logic (e.g. to communicate with a theorem prover)
- A logic can be expressive enough to state a property but not expressive enough to make the proof
- For example the logic \( L_0 = \{\text{true, false}\} \) can express non-termination
  \( \{\text{true}\} \text{ while B \ fo C \ od \{false\}} \)
- Obviously proving non-termination requires a loop invariant which is neither true nor false so the proof cannot be done with this logic \( L_0 \)
Beyond invariance
(facultative home work)

Invariance from intermediate states

• Sometimes we want to consider an invariant for execution traces starting at the reachable states of the semantics from $S$, satisfying a given condition $P$.

Invariance from intermediate states

• The proof method for the intermediate invariant $J$ is the same for initial states given as the intersection of $P$ with an over-approximation $I$ of the reachable states of $S$ from $S$:
  
  • Find $I$ and prove it to be an invariant of $S$ from initial states $S$.
  
  • Prove $I \cap P \subseteq J$ (i.e. $J$ holds for its intermediate initial states).
  
  • Prove that for all states $s, s'$, if $s \in J$ and $s'$ is a possible successor of $s$ (i.e. $\tau(s, s')$ is true) then $s' \in J$.

Observational equivalence
of execution traces
(facultative home work)
Semantic observational equivalence

- Two trace semantics \( S, S' \in \wp(T) \) are observationally equivalent for an abstraction \( \alpha \in \wp(T) \mapsto \wp(O) \) in observations \( O \), if and only if, by definition,
  \[ \alpha(S) = \alpha(S') \]

- This is written \( S \equiv_{\alpha} S' \)

- Observe that we may have \( S \equiv_{\alpha} S' \) that is \( \alpha(S) = \alpha(S') \) even though \( S \neq S' \)

Program observational equivalence

- Two programs \( P \) and \( P' \) are observationally equivalent for an abstraction if and only if, by definition, their respective trace semantics are observationally equivalent for that abstraction.

Example

- The following two semantics are observationally equivalent for the pre-post abstraction:

\[
S = \begin{cases}
1: \text{int } x; \\
2: \text{input}(x); \\
3: x := x+2; \\
4: \text{.../...}
\end{cases}
\]

\[
S' = \begin{cases}
1: \text{int } x; \\
2: \text{input}(x); \\
3: x := x+1; \\
4: x := x+1; \\
5: \text{.../...}
\end{cases}
\]

Example

- The following two programs \( P \) and \( P' \) are observationally equivalent for the pre-post abstraction:

\[
P =
\begin{align*}
1: & \text{int } x; \\
2: & \text{input}(x); \\
3: & x := x+2; \\
4: & \text{.../...}
\end{align*}
\]

\[
P' =
\begin{align*}
1: & \text{int } x; \\
2: & \text{input}(x); \\
3: & x := x+1; \\
4: & x := x+1; \\
5: & \text{.../...}
\end{align*}
\]
Observational equivalence for pre-post properties ($\alpha$)

- $\text{skip ; C } \equiv_{\alpha} \text{ C ; skip } \equiv_{\alpha} \text{ C}$
- $\text{x := x } \equiv_{\alpha} \text{ skip}$
- $\text{x := f(x) ; x := g(x) } \equiv_{\alpha} \text{ x := g(f(x))}$
- $\text{if true C else C' } \equiv_{\alpha} \text{ C}$
- $\text{if false C else C' } \equiv_{\alpha} \text{ C'}$
- $\text{while B ( C ) } \equiv_{\alpha} \text{ if B ( C ; while B ( C ) ) else skip}$
- etc...

$f, g, \text{ and } B$ are assumed to have no side-effect (do not modify any variable)

Conclusion

- Formal methods aim at rigorous mathematical reasoning on the behavior of programs at runtime (semantics)
- They can be automated (theorem provers, SMT solvers, abstract interpreters, model-checkers, …)
- They are the foundations of provable software security e.g.
  - Invariance proofs: absence of buffer overruns
  - Equivalent proof: program $\equiv$ program+attacker $\Longrightarrow$ no observable attack!

Homework (facultative)
Invariance proof of a program

- Complete the missing local invariants and prove the following linear search program correct:

```c
1: /* n = n0 ≥ 1 ∧ ∃k ∈ [1,n]: A[k] = A0[k] ∈ ℤ ∧ x = x0 ∈ ℤ */
   i := 1.
2: /* … */
while (i <= n) && (A[i] != x)
{ 3: /* … */
   i := i + 1;
4: /* … */
}
5: /* n = n0 ∧ ∃k ∈ [1,n]: A[k] = A0[k] ∧ x = x0 ∧
   ((1≤i≤n ∧ A[i] = x) ∨ (i = n+1 ∧ ∃k ∈ [1,n]: A[k] ≠ x)) */
```

Historical References

- Peter Naur, Proof of algorithms by general snapshots, BIT Numerical Mathematics Volume 6, Number 4 (1966), 310-316

Reference

- A proof that invariance (and termination) proof methods are abstract interpretations of the program finite and infinite trace (the fundamental idea underlying this class):

Patrick Cousot, Radhia Cousot: An abstract interpretation framework for termination, POPL 2012: 245-258
The End