Deductive Verification of Programs

6.0 VU Formal Methods in Computer Science

Gernot Salzer

AB Theoretische Informatik und Logik
Institut für Computersprachen

18 November 2013

Topics last time

1. Why formal methods?
2. Syntax of TPL (Toy Programming Language)
3. Operational Semantics of TPL

Why formal methods?

Formal (= mathematical) methods ...

- allow us to guarantee (= prove) properties of programs/systems;
- are necessary if the computer is expected to help us;
- improve the quality of software by enforcing rigorous and structured thinking.

Formal proofs require that all involved parts have been formalised before.
Parts: specification, semantics of program, machine architecture, ...

This part of the course ...

- concentrates on functional requirements of sequential programs.
- uses methods of deductive verification.
- introduces
  - operational semantics to define the meaning of imperative programs and
  - axiomatic semantics to verify properties of imperative programs.

Toy Programming Language (TPL)

$$
\begin{array}{l}
z := 0; \\
\textbf{while } y \neq 0 \textbf{ do} \\
\quad z := z + x; \\
\quad y := y - 1 \\
\textbf{od}
\end{array}
$$

- Which character strings can be interpreted as programs?
  $\Longrightarrow$ syntax of TPL
- What do programs mean?
  $\Longrightarrow$ semantics of TPL
- What is the program supposed to do?
  $\Longrightarrow$ formal specification of intended behaviour
- Does the program do what it is supposed to do?
  $\Longrightarrow$ formal verification of program

Topics last time

1. Why formal methods?
2. Syntax of TPL (Toy Programming Language)
3. Operational Semantics of TPL

TPL Syntax

$\mathcal{P}$ ::= “**skip**” | “**abort**” | $\mathcal{V}$ “:=” $\mathcal{E}$ | $\mathcal{P}$ “;” $\mathcal{P}$ — programs
| “**if**” $\mathcal{E}$ “**then**” $\mathcal{P}$ “**else**” $\mathcal{P}$ “**fi**”
| “**while**” $\mathcal{E}$ “**do**” $\mathcal{P}$ “**od**”

$\mathcal{E}$ ::= $\mathcal{V}$ | $\mathcal{N}$ | $\mathcal{U}\,\mathcal{E}$ | “(” $\mathcal{E}\,\mathcal{B}\,\mathcal{E}$ “)” — expressions

$\mathcal{V}$ ::= “x” | “y” | $\cdots$ | any word except key words | $\cdots$ — variables

$\mathcal{N}$ ::= “0” | “1” | $\cdots$ | “9” | “10” | “11” | $\cdots$ — numerals

$\mathcal{U}$ ::= “$+$” | “$-$” | “$\neg$” | $\cdots$ — unary operators

$\mathcal{B}$ ::= “$+$” | “$-$” | “$*$” | “$/$” | “$<$” | “$\leq$” | “$=$” | $\cdots$ — binary operators

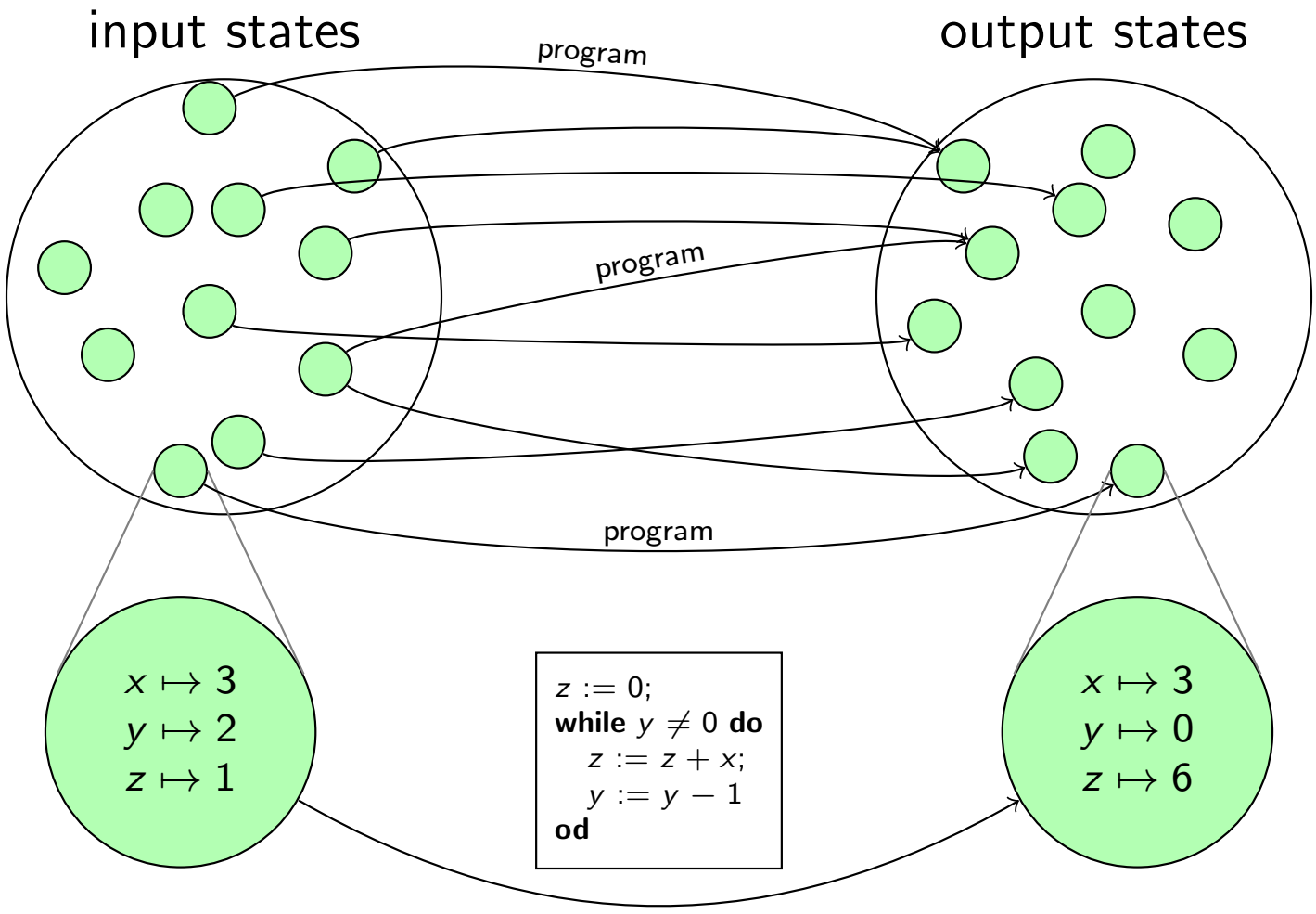
Overloading: $\mathcal{P}$, $\mathcal{E}$, $\mathcal{V}$, $\mathcal{N}$, $\mathcal{U}$, $\mathcal{B}$ denote

- grammar variables
- the languages generated from these variables

Topics last time

1. Why formal methods?
2. Syntax of TPL(toy programming language)
3. Operational Semantics of TPL

Programs as State Transformers



Program state:

- Informally: memory snapshot
- Formally: mapping from variables to values, $\mathcal{V} \mapsto \mathbb{Z}$
- Set of all states: $\mathcal{S} \stackrel{\text{def}}{=} \{\, \sigma \mid \sigma : \mathcal{V} \mapsto \mathbb{Z} \,\}$

Configuration:

- Informally: system snapshot ("dump") = remaining program + state
- Formally: pair (p, σ) with program $p \in \mathcal{P}$ and state $\sigma \in \mathcal{S}$, or just a state σ (final configuration).
- Set of all configurations: $\mathcal{C} \stackrel{\text{def}}{=} (\mathcal{P} \times \mathcal{S}) \cup \mathcal{S}$

Transition relation:

- Informally: describes a single computation step
- Formally: relation $\Rightarrow \subseteq (\mathcal{P} \times \mathcal{S}) \times \mathcal{C}$
- Program run: sequence of transitions

Transition Relation for TPL

$$(\textbf{skip}, \sigma) \Rightarrow \sigma$$

$$(v := e, \sigma) \Rightarrow \sigma' \quad \text{where} \quad \begin{array}{l} \sigma'(v) = [e]\,\sigma \\ \sigma'(x) = \sigma(x) \quad \text{for } x \neq v \end{array}$$

$$(p; q, \sigma) \Rightarrow \begin{cases} (p'; q, \sigma') & \text{if } (p, \sigma) \Rightarrow (p', \sigma') \\ (q, \sigma') & \text{if } (p, \sigma) \Rightarrow \sigma' \end{cases}$$

$$(\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}, \sigma) \Rightarrow \begin{cases} (p, \sigma) & \text{if } [e]\,\sigma \neq 0 \\ (q, \sigma) & \text{if } [e]\,\sigma = 0 \end{cases}$$

$$(\textbf{while } e \textbf{ do } p \textbf{ od}, \sigma) \Rightarrow \begin{cases} (p; \textbf{while } e \textbf{ do } p \textbf{ od}, \sigma) & \text{if } [e]\,\sigma \neq 0 \\ \sigma & \text{if } [e]\,\sigma = 0 \end{cases}$$

- Note that **abort** is missing.
- Kind of abstract TPL interpreter.

Semantics of Expressions

$[e]\colon \mathcal{S} \mapsto \mathbb{Z}$ ... function computed by expression $e \in \mathcal{E}$

$[v]\,\sigma = \sigma(v)$ for $v \in \mathcal{V}$

$\sigma(v)$ ... value of v in current state σ

$[n]\,\sigma = [n]$ for $n \in \mathcal{N}$

$[n] \in \mathbb{Z}$ ... integer corresponding to numeral n
$[0] = 0$, $[1] = 1$, $[2] = 2$, ...

$[u\,e]\,\sigma = [u]([e]\,\sigma)$ for $u \in \mathcal{U}$

$[u]\colon \mathbb{Z} \mapsto \mathbb{Z}$ ... unary function corresponding to operator u
$[\neg]$ = boolean negation, $[-]$ = unary minus

$[e\,b\,e']\,\sigma = [b]([e]\,\sigma, [e']\,\sigma)$ for $b \in \mathcal{B}$

$[b]\colon \mathbb{Z}^2 \mapsto \mathbb{Z}$ ... binary function corresponding to operator b
$[\wedge]$, $[\vee]$, $[\Rightarrow]$ ... binary boolean functions
$[+]$, $[-]$, $[*]$, $[/]$ ... binary integer functions
$[<]$, $[\leq]$, $[=]$, $[\geq]$, $[>]$ ... comparison of integers

Structural Operational Semantics (SOS) of TPL

The function $[p]\colon \mathcal{S} \mapsto \mathcal{S}$ computed by a program p is defined by

$[p]\sigma = \sigma'$ if and only if $(p, \sigma) \stackrel{*}{\Rightarrow} \sigma'$ for all states $\sigma, \sigma' \in \mathcal{S}$.

($\stackrel{*}{\Rightarrow}$ ... reflexive and transitive closure of $\Rightarrow$)

Semantic equivalence

Two programs p and q are semantically equivalent if $[p] = [q]$.

This means:

- If $(p, \sigma) \stackrel{*}{\Rightarrow} \sigma'$, then $(q, \sigma) \stackrel{*}{\Rightarrow} \sigma'$, and vice versa.
- If (p, σ) loops or aborts, then so does (q, σ), and vice versa.

Note: The semantic function $[p]$ does not distinguish between endless loops and abortion, even though the transition relation does.

Topics today

1. Why formal methods?
2. Syntax of TPL(toy programming language)
3. Operational Semantics of TPL
4. Correctness assertions
5. How to prove correctness assertions

Theorem

For all progams p, q, variables v, expressions e and all states σ, the SOS of TPL has the following properties:

- $[\textbf{skip}]\,\sigma = \sigma$
- $[v := e]\,\sigma = \sigma'$, where $\begin{aligned} &\sigma'(v) = [e]\,\sigma \\ &\sigma'(x) = \sigma(x) \quad \text{for } x \neq v \end{aligned}$
- $[p; q]\,\sigma = [q]\,[p]\,\sigma$
- $[\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}]\,\sigma = \begin{cases} [p]\,\sigma & \text{if } [e]\,\sigma \neq 0 \\ [q]\,\sigma & \text{if } [e]\,\sigma = 0 \end{cases}$
- $[\textbf{while } e \textbf{ do } p \textbf{ od}]\,\sigma = \begin{cases} [\textbf{while } e \textbf{ do } p \textbf{ od}]\,[p]\,\sigma & \text{if } [e]\,\sigma \neq 0 \\ \sigma & \text{if } [e]\,\sigma = 0 \end{cases}$

$$[\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}]\,\sigma = \begin{cases} [p]\,\sigma & \text{if } [e]\,\sigma \neq 0 \\ [q]\,\sigma & \text{if } [e]\,\sigma = 0 \end{cases}$$

Proof:

$[\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}]\,\sigma = \sigma'$

$$\iff (\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}, \sigma) \stackrel{*}{\Rightarrow} \sigma' \qquad \text{Def. of } [p]$$

$$\text{SOS: } (\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}, \sigma) \Rightarrow \begin{cases} (p, \sigma) & \text{if } [e]\,\sigma \neq 0 \\ (q, \sigma) & \text{if } [e]\,\sigma = 0 \end{cases}$$

$$\iff \begin{cases} (p, \sigma) \stackrel{*}{\Rightarrow} \sigma' & \text{if } [e]\,\sigma \neq 0 \\ (q, \sigma) \stackrel{*}{\Rightarrow} \sigma' & \text{if } [e]\,\sigma = 0 \end{cases} \qquad \text{Def. SOS of } \textbf{if}$$

$$\iff \begin{cases} [p]\,\sigma = \sigma' & \text{if } [e]\,\sigma \neq 0 \\ [q]\,\sigma = \sigma' & \text{if } [e]\,\sigma = 0 \end{cases} \qquad \text{Def. of } [p]$$

Sequential composition is associative

$[(p_1; p_2); p_3] = [p_1; (p_2; p_3)]$ for all programs p_1, p_2, and p_3.

Proof: Let τ be an arbitrary state. We have:

$$
\begin{aligned}
[(p_1; p_2); p_3]\,\tau &= [p_3]\,([p_1; p_2]\,\tau) && \text{Theorem: } [p; q]\,\sigma = [q]\,[p]\,\sigma \\
&= [p_3]\,([p_2]\,([p_1]\,\tau)) && \text{Theorem: } [p; q]\,\sigma = [q]\,[p]\,\sigma \\
&= [p_2; p_3]\,([p_1]\,\tau) && \text{Theorem: } [q]\,[p]\,\sigma = [p; q]\,\sigma \\
&= [p_1; (p_2; p_3)]\,\tau && \text{Theorem: } [q]\,[p]\,\sigma = [p; q]\,\sigma
\end{aligned}
$$

This result allows us to evaluate the first statement in a sequential composition, regardless of its tree structure.

Natural Semantics of TPL

Idea: Use theorem as definition of $[p] : \mathcal{S} \mapsto \mathcal{S}$

Definition

For all progams p, q, variables v, expressions e and all states σ, the natural semantics of TPL is defined by:

- $[\textbf{skip}]\,\sigma \stackrel{\text{def}}{=} \sigma$
- $[v := e]\,\sigma \stackrel{\text{def}}{=} \sigma'$, where $\begin{array}{l} \sigma'(v) = [e]\,\sigma \\ \sigma'(x) = \sigma(x) \quad \text{for } x \neq v \end{array}$
- $[p; q]\,\sigma \stackrel{\text{def}}{=} [q]\,[p]\,\sigma$
- $[\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}]\,\sigma \stackrel{\text{def}}{=} \begin{cases} [p]\,\sigma & \text{if } [e]\,\sigma \neq 0 \\ [q]\,\sigma & \text{if } [e]\,\sigma = 0 \end{cases}$
- $[\textbf{while } e \textbf{ do } p \textbf{ od}]\,\sigma \stackrel{\text{def}}{=} \begin{cases} [\textbf{while } e \textbf{ do } p \textbf{ od}]\,[p]\,\sigma & \text{if } [e]\,\sigma \neq 0 \\ \sigma & \text{if } [e]\,\sigma = 0 \end{cases}$

Natural Semantics: Example

$$
\begin{aligned}
[p]\,\sigma &= [z := 0; \textbf{while } \ldots]\,\sigma \\
&= [\textbf{while } \ldots]\,[z := 0]\,\sigma \\
&= [\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}]\,\sigma_1 \\
&= [\textbf{while } \ldots]\,[z := z + x; y := y - 1]\,\sigma_1 \\
&= [\textbf{while } \ldots]\,[y := y - 1]\,[z := z + x]\,\sigma_1 \\
&= [\textbf{while } \ldots]\,[y := y - 1]\,\sigma_2 \\
&= [\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}]\,\sigma_3 \\
&= [\textbf{while } \ldots]\,[z := z + x; y := y - 1]\,\sigma_3 \\
&= [\textbf{while } \ldots]\,[y := y - 1]\,[z := z + x]\,\sigma_3 \\
&= [\textbf{while } \ldots]\,[y := y - 1]\,\sigma_4 \\
&= [\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}]\,\sigma_5 \\
&= \sigma_5
\end{aligned}
$$

p: $z := 0;$
while $y \neq 0$ **do**
$z := z + x;$
$y := y - 1$
od

σ: $x \mapsto 3, y \mapsto 2, z \mapsto 1$

σ_1: $z \mapsto [0]\,\sigma = 0$
$x \mapsto 3, y \mapsto 2$

$[y \neq 0]\,\sigma_1 = 1$ (true)

σ_2: $z \mapsto [z + x]\,\sigma_1 = 3$
$x \mapsto 3, y \mapsto 2$

σ_3: $y \mapsto [y - 1]\,\sigma_2 = 1$
$x \mapsto 3, z \mapsto 3$

$[y \neq 0]\,\sigma_3 = 1$ (true)

σ_4: $z \mapsto [z + x]\,\sigma_3 = 6$
$x \mapsto 3, y \mapsto 1$

σ_5: $y \mapsto [y - 1]\,\sigma_4 = 0$
$x \mapsto 3, z \mapsto 6$

$[y \neq 0]\,\sigma_5 = 0$ (false)

Compare to SOS:

$$(p,\sigma) = (z := 0; \textbf{while } \ldots, \sigma)$$
$$(z := 0, \sigma) \Rightarrow \sigma_1$$
$$\Rightarrow (\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}, \sigma_1)$$
$$\Rightarrow (z := z + x; y := y - 1; \textbf{while } \ldots, \sigma_1)$$
$$(z := z + x; y := y - 1, \sigma_1)$$
$$(z := z + x, \sigma_1) \Rightarrow \sigma_2$$
$$\Rightarrow (y := y - 1, \sigma_2)$$
$$\Rightarrow (y := y - 1; \textbf{while } \ldots, \sigma_2)$$
$$(y := y - 1, \sigma_2) \Rightarrow \sigma_3$$
$$\Rightarrow (\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}, \sigma_3)$$
$$\Rightarrow (z := z + x; y := y - 1; \textbf{while } \ldots, \sigma_3)$$
$$(z := z + x; y := y - 1, \sigma_3)$$
$$(z := z + x, \sigma_3) \Rightarrow \sigma_4$$
$$\Rightarrow (y := y - 1, \sigma_4)$$
$$\Rightarrow (y := y - 1; \textbf{while } \ldots, \sigma_4)$$
$$(y := y - 1, \sigma_4) \Rightarrow \sigma_5$$
$$\Rightarrow (\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}, \sigma_5)$$
$$\Rightarrow \sigma_5$$

p: $z := 0;$
while $y \neq 0$ **do**
$z := z + x;$
$y := y - 1$
od

σ: $x \mapsto 3, y \mapsto 2, z \mapsto 1$

σ_1: $z \mapsto [0]\sigma = 0$
$x \mapsto 3, y \mapsto 2$

$[y \neq 0]\sigma_1 = 1$ (true)

σ_2: $z \mapsto [z + x]\sigma_1 = 3$
$x \mapsto 3, y \mapsto 2$

σ_3: $y \mapsto [y - 1]\sigma_2 = 1$
$x \mapsto 3, z \mapsto 3$

$[y \neq 0]\sigma_3 = 1$ (true)

σ_4: $z \mapsto [z + x]\sigma_3 = 6$
$x \mapsto 3, y \mapsto 1$

σ_5: $y \mapsto [y - 1]\sigma_4 = 0$
$x \mapsto 3, z \mapsto 6$

$[y \neq 0]\sigma_5 = 0$ (false)

Structural operational vs. natural semantics

Natural semantics:

- no transitions, no program runs
- just a recursive definition relating input to output states
- more elegant, easier to use, more compact

Structural operational semantics:

- distinguishes between infinite loops and abortion
- allows us to model fine-grained parallelism properly

Topics today

1. Why formal methods?
2. Syntax of TPL(toy programming language)
3. Operational Semantics of TPL
4. Correctness assertions
5. How to prove correctness assertions

Correctness Assertions

Correctness assertion: "$\mathcal{S}_{\text{in}}\ p\ \mathcal{S}_{\text{out}}$"
"Program p transforms the states in $\mathcal{S}_{\text{in}}$ to states in $\mathcal{S}_{\text{out}}$."

May be true or false $\Longrightarrow$ kind of logical formula

What about inputs with undefined outputs?

Assertion is true w.r.t. partial correctness (is "partially correct"/p.c.) if:
Whenever the input state is in $\mathcal{S}_{\text{in}}$ and the program terminates,
then the output state is in $\mathcal{S}_{\text{out}}$.

Assertion is true w.r.t. total correctness (is "totally correct"/t.c.) if:
Whenever the input state is in $\mathcal{S}_{\text{in}}$,
then the program terminates and the output state is in $\mathcal{S}_{\text{out}}$.

total correctness = partial correctness + termination

1. $\{\,(3,2,1)\,\}\,p\,\{\,(3,0,6)\,\}$
   p.c. and t.c.
2. $\{\,(3,2,1),(3,2,0)\,\}\,p\,\{\,(3,0,6)\,\}$
   p.c. and t.c.
3. $\{\,(3,2,1),(3,-1,0)\,\}\,p\,\{\,(3,0,6)\,\}$
   p.c. but not t.c.
4. $\{\,(3,2,1)\,\}\,p\,\{\,(3,0,6),(0,0,0)\,\}$
   p.c. and t.c.
5. $\{\,(3,3,3)\,\}\,p\,\{\,(3,0,6)\,\}$
   neither p.c. nor t.c.
6. $\{\,(3,3,3)\,\}\,p\,\{\,(3,0,9)\,\}$
   p.c. and t.c.
7. $\{\,(3,2,1),(3,3,3)\,\}\,p\,\{\,(3,0,6),(3,0,9)\,\}$
   p.c. and t.c.

(a,b,c): $x \mapsto a$
$y \mapsto b$
$z \mapsto c$

p: $z := 0$;
while $y \neq 0$ **do**
$z := z + x$;
$y := y - 1$
od

Correlating Input and Output States

Single test cases can be combined:

- $\{(3,2,1)\}\, p\, \{(3,0,6)\}$ is correct.
- $\{(3,3,3)\}\, p\, \{(3,0,9)\}$ is correct.

Therefore:

- $\{(3,2,1),(3,3,3)\}\, p\, \{(3,0,6),(3,0,9)\}$ is correct.

The opposite, however, is not true.
Which inputs correspond to which outputs?

Problem: We need the opposite.

- Prove a combined correctness assertion with big sets.

And we want to conclude:

- Each single test case is correct.

Solution: augment states with auxiliary variables.
(Also called "logical variables", in contrast to program variables.)

- Input states are characterized by x and y.
- Output states miss the original value of y.
- Therefore we add the auxiliary variable y_0 that contains the original value of y.

Now we can reverse the argument:

- $\{ \begin{matrix}(3,2,1,2),\\(3,3,3,3)\end{matrix} \}\, p\, \{ \begin{matrix}(3,0,6,2),\\(3,0,9,3)\end{matrix} \}$ is correct.

Therefore:

- $\{\,(3,2,1,2)\,\}\, p\, \{\,(3,0,6,2)\,\}$ is correct.
- $\{\,(3,3,3,3)\,\}\, p\, \{\,(3,0,9,3)\,\}$ is correct.

(a,b,c,d): $x \mapsto a$
$y \mapsto b$
$z \mapsto c$
$y_0 \mapsto d$

p: $z := 0;$
while $y \neq 0$ **do**
$z := z + x;$
$y := y - 1$
od

Beyond Testing: Infinite Sets of States

How to prove assertions with infinite sets of input/output states?

- Input states: $\mathcal{S}_{\text{in}} = \{\, (a, b, c, b) \mid a, b, c \in \mathbb{Z} \,\}$
- Output states: $\mathcal{S}_{\text{out}} = \{\, (a, 0, a \cdot d, d) \mid a, d \in \mathbb{Z} \,\}$

Proving $\mathcal{S}_{\text{in}}\ p\ \mathcal{S}_{\text{out}}$ amounts to infinitely many test cases.

We need:

- a language to specify infinite sets of states
  $\Longrightarrow$ first-order logic
- a method to handle assertions with infinite sets of states
  $\Longrightarrow$ Hoare calculus & friends:
  - Hoare calculus
  - weakest (liberal) preconditions
  - strongest postconditions
  - annotation calculus
    practical tool using all of the above

First-Order Formulas

Syntax: $\mathcal{F} ::= \mathcal{V} \mid \mathcal{N} \mid \mathcal{U}\,\mathcal{F} \mid \text{“(”}\ \mathcal{F}\,\mathcal{B}\,\mathcal{F}\ \text{“)”} \mid \text{“}\forall\text{”}\,\mathcal{V}\,\mathcal{F} \mid \text{“}\exists\text{”}\,\mathcal{V}\,\mathcal{F}$

Semantics: The same as for $\mathcal{E}$, except:

$$[\forall v\ e]\,\sigma = \begin{cases} 1 & \text{if } [e]\,\sigma' \neq 0 \text{ for all } \sigma' \in \mathcal{S} \text{ such that } \sigma' \overset{v}{\sim} \sigma \\ 0 & \text{otherwise} \end{cases}$$

$$[\exists v\ e]\,\sigma = \begin{cases} 1 & \text{if } [e]\,\sigma' \neq 0 \text{ for some } \sigma' \in \mathcal{S} \text{ such that } \sigma' \overset{v}{\sim} \sigma \\ 0 & \text{otherwise} \end{cases}$$

$\sigma' \overset{v}{\sim} \sigma$: $\sigma'(x) = \sigma(x)$ for all variables $x \neq v$

“States σ and σ' differ at most at variable v.”

Is $[\forall x\, \exists y\, y > x]\, \sigma$ true in an arbitrary state σ?

$[\forall x\, \exists y\, y > x]\, \sigma = 1$

if $[\exists y\, y > x]\, \sigma' = 1$ for all $\sigma' \overset{x}{\sim} \sigma$

(i.e., $\sigma'(x) = n$ for an arbitrary $n \in \mathbb{Z}$)

if $[y > x]\, \sigma'' = 1$ for all $\sigma' \overset{x}{\sim} \sigma$ and some $\sigma'' \overset{y}{\sim} \sigma'$

(i.e., $\sigma''(x) = n$, and we choose $\sigma''(y) = n + 1$)

$$\begin{aligned}
[y > x]\, \sigma'' &= [>]([y]\, \sigma'', [x]\, \sigma'') \\
&= (\sigma''(y) > \sigma''(x)) \\
&= (n + 1 > n) \\
&= 1
\end{aligned}$$

Therefore the formula is valid.

Formulas vs. Sets of States

$\{ F \}$ ... set of states defined by formula F "F-states"

$$\{ F \} \stackrel{\text{def}}{=} \{ \sigma \in \mathcal{S} \mid [F]\sigma \neq 0 \}$$

- $\mathcal{S}_{\text{in}} = \{ (a, b, c, b) \mid a, b, c \in \mathbb{Z} \} = \{ y = y_0 \}$
- $\mathcal{S}_{\text{out}} = \{ (a, 0, a \cdot d, d) \mid a, d \in \mathbb{Z} \} = \{ y = 0 \wedge z = x * y_0 \}$

Remember: (a, b, c, d) means $x \mapsto a$, $y \mapsto b$, $z \mapsto c$, and $y_0 \mapsto d$.

Some observations:

- $\{ F \wedge G \} = \{ F \} \cap \{ G \}$
- $\{ F \vee G \} = \{ F \} \cup \{ G \}$
- $F \Rightarrow G$ is valid iff $\{ F \} \subseteq \{ G \}$

Can we define every set of states by a formula?

Correctness Assertions and Formulas

$\{ F \}\, p \,\{ G \}$ is true regarding partial correctness (is "partially correct"), if

- ... whenever p starts in an F-state and terminates,
  then p stops in a G-state.
- ... for all states $\sigma \in \mathcal{S}$,
  if $[F]\,\sigma$ is true and $[p]\,\sigma$ is defined,
  then $[G]\,[p]\,\sigma$ is true.

$\{ F \}\, p \,\{ G \}$ is true regarding total correctness (is "totally correct"), if

- ... whenever p starts in an F-state,
  then p terminates and stops in a G-state.
- ... for all states $\sigma \in \mathcal{S}$,
  if $[F]\,\sigma$ is true,
  then $[p]\,\sigma$ is defined and $[G]\,[p]\,\sigma$ is true.

F ... precondition
G ... postcondition

$\{\,1\,\}\, x := 2 \,\{\, x = 2 \,\}$

Totally (and therefore also partially) correct.

$\{\,1\,\}\, x := 2 \,\{\, x = 3 \,\}$

Neither totally nor partially correct.

$\{\,1\,\}$ **while** $x > 2$ **do** $x := x - 1 \,\{\, x = 2 \,\}$

Neither totally nor partially correct. Counterexample: $\sigma(x) = 0$

$\{\,1\,\}$ **while** $x \neq 2$ **do** $x := x - 1 \,\{\, x = 2 \,\}$

Partially but not totally correct. Counterexample: $\sigma(x) = 0$

$\{\, x > 5 \,\}$ **while** $x \neq 2$ **do** $x := x - 1 \,\{\, x = 2 \,\}$

Totally (and therefore also partially) correct.

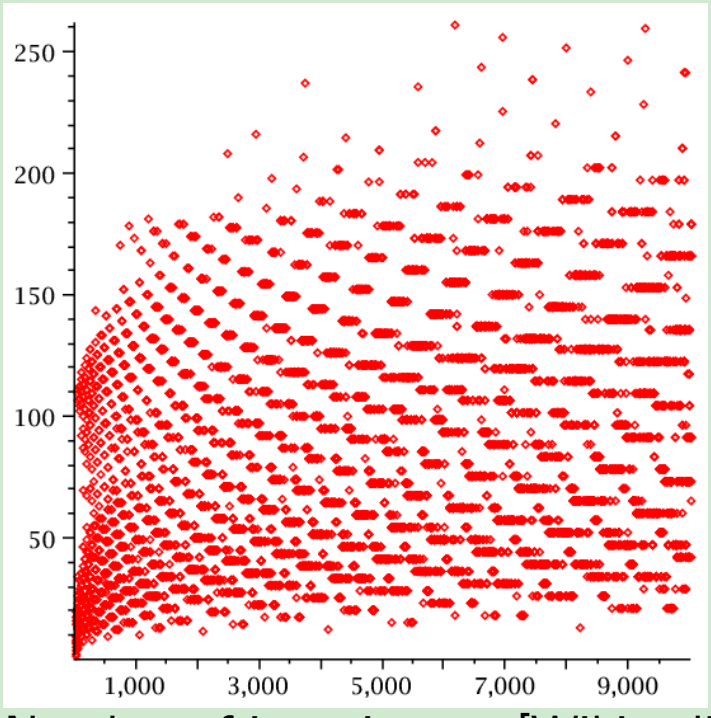
$\{ y = y_0 \}$
$z := 0;$
while $y \neq 0$ **do**
  $z := z + x;$
  $y := y - 1$
od
$\{ z = x * y_0 \}$

Partially correct? It seems so.

Terminating? No, not for $y < 0$.
$\Longrightarrow$ Add $y \geq 0$ to the precondition.

But how to prove it?
We have to check infinitely many input states. Infeasible.
$\Longrightarrow$ Hoare calculus to the rescue

$\{ x \geq 1 \}$
while $x > 1$ **do**
  if $x = 2 * (x/2)$ **then**
    $x := x/2$
  else
    $x := 3 * x + 1$
  fi
od
$\{ x = 1 \}$



Number of iterations [Wikipedia]

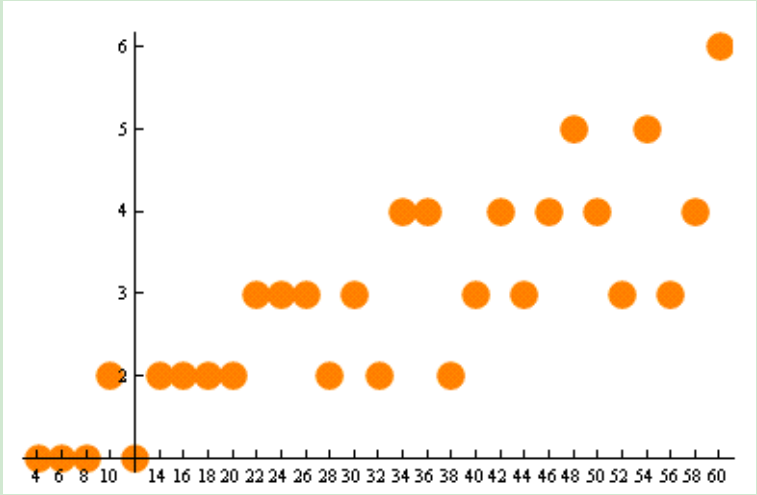
Partially correct? Obviously.
x is always positive.
On termination we have $x \not> 1$, hence x must be equal to 1.

Terminating? Yes, if $x_n = 1$ for some n, for every $x_0 \geq 1$.

$$x_{n+1} = \begin{cases} x_n/2 & \text{if } x_n \text{ even} \\ 3x_n + 1 & \text{if } x_n \text{ odd} \end{cases}$$

Collatz conjecture, open problem

$\{ x \geq 2 \}$
$y := 2$
while $y < x$
$\quad \wedge \neg(\text{prime}(y) \wedge \text{prime}(2x - y))$ **do**
$\quad y := y + 1$
od
$\{ \text{prime}(y) \wedge \text{prime}(2x - y) \}$



Number of decompositions [Wikip.]

Partially correct?
Yes, if every even integer greater than 2 is the sum of two primes.
Goldbach conjecture, notorious open problem.

Terminating? Obviously, at most $x - 2$ iterations.

Conclusion:

- Some assertions will be hard for every verification tool.
- Fortunately, assertions in practise are much easier.

Topics today

1. Why formal methods?
2. Syntax of TPL(toy programming language)
3. Operational Semantics of TPL
4. Correctness assertions
5. How to prove correctness assertions

Weak and strong formulas

F, G ... formulas

F is weaker than G, if F is implied by G ($G \Rightarrow F$ is valid).
F is stronger than G, if F implies G ($F \Rightarrow G$ is valid).

"weaker" = "less restrictive" = "more satisfying states"
"stronger" = "more restrictive" = "less satisfying states"

$F \Rightarrow G$ is valid if and only if $\{ F \} \subseteq \{ G \}$.

$x = y$ is stronger than $x \geq y$.

$x > y$ is weaker than $x = y + 1$.

1 (true) is the weakest formula: implied by everything, $\{ 1 \} = \mathcal{S}$.

0 (false) is the strongest formula: implies everything, $\{ 0 \} = \emptyset$.

$x = 2$ and $x > y$ are incomparable: neither is weaker than the other.

Three ways to prove correctness assertions

Task: Show that $\{ F \}\, p \,\{ G \}$ is partially/totally correct.

Method 0: Use the definition.

Show that for all states $\sigma \in \mathcal{S}$ satisfying the precondition F, the state after executing program p, $[p]\,\sigma$, satisfies the postcondition G.

Problem: $\mathcal{S}$ infinite, too many states to check.

Method 1: Hoare calculus.

Decompose correctness assertion into simpler ones (guided by rules) until we obtain true assertions (instances of axioms) and valid formulas.

Method 2: Weakest (liberal) precondition.

Compute the weakest formula F' such that $\{ F' \}\, p \,\{ G \}$ is true, and show that F implies F'.

Method 3: Strongest postcondition.

Compute the strongest formula G' such that $\{ F \}\, p \,\{ G' \}$ is true, and show that G' implies G.

Method 4: Annotation calculus.

No new method, just combines the above methods for practical usage.