Exercise 06 - Solution

Interval Analysis

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) (i) Define multiplication $*^i \colon L^i \times L^i \to L^i$ as follows. If any of the two arguments is $\bot_i$, the result is $\bot_i$. Otherwise:

$$[a, b] *^i [c, d] \;=\; [\min(ac, ad, bc, bd),\ \max(ac, ad, bc, bd)]$$

Note: To see this, perform a case distinction on the signs of a, b, c, and d.

(ii) Define $|\cdot|^i \colon L^i \to L^i$ as follows.

$$|\bot_i| = \bot_i$$

$$|[a, b]| = \begin{cases} [-b, -a] & \text{if } b \leq 0, \\ [a, b] & \text{else if } a \geq 0, \\ [0, \max(-a, b)] & \text{otherwise.} \end{cases}$$

Note: Consider the 3 possible positions of the interval with respect to 0.

(iii) The equality constraint `x == y` results in the intersection of the intervals for `x` and `y` using the meet operation $\sqcap_i$.

Formally, the transformer $[\![\texttt{x == y}]\!]_i$ is defined as follows. For $m \colon \text{Vars} \to L^i$:

$$[\![\texttt{x == y}]\!]_i(m) = m[\texttt{x} \mapsto m(\texttt{x}) \sqcap_i m(\texttt{y}),\ \texttt{y} \mapsto m(\texttt{x}) \sqcap_i m(\texttt{y})]$$

b) The entries l_x, l_y in the following table describe the state updates to `x` and `y` for the constraint `x < y`. Empty entries mean that the state remains unchanged.

		y: $+$	y: $-$	y: 0	y: $\top$	y: $\bot$
x	$+$		$\bot, \bot$	$\bot, \bot$	$+, +$	$\bot, \bot$
x	$-$					$\bot, \bot$
x	0		$\bot, \bot$	$\bot, \bot$	$0, +$	$\bot, \bot$
x	$\top$		$-, -$	$-, 0$		$\bot, \bot$
x	$\bot$	$\bot, \bot$	$\bot, \bot$	$\bot, \bot$	$\bot, \bot$	$\bot, \bot$

Note: Most cases either lead to unsatisfiable constraints $(\bot, \bot)$ or don't allow us to learn anything new. Still, four cases (see highlighted) are interesting. For example, the constraint $\top < -$ allows us to conclude that x must be negative.

Solution 2.

a) Abstract the input by $[1, \infty]$. While this is not exact (it contains also even numbers), it is the most precise interval approximation.

b) First, we introduce phantom labels $3'$ and $4'$ pointing at the state immediately after line 3, resp. line 4. This simplifies joining the two branches in line 5.

The iterations are shown below. Note how for the first line, we initialize the state of all variables to $[-\infty, \infty]$, except for x, where we use the interval from (a). Also, note the join operation involved in the fourth iteration ($\overset{*}{\rightsquigarrow}$).

	x	y	z
1	$[1, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$\bot$	$\bot$	$\bot$
3	$\bot$	$\bot$	$\bot$
$3'$	$\bot$	$\bot$	$\bot$
4	$\bot$	$\bot$	$\bot$
$4'$	$\bot$	$\bot$	$\bot$
5	$\bot$	$\bot$	$\bot$
6	$\bot$	$\bot$	$\bot$

$\rightsquigarrow$

	x	y	z
1	$[1, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[1, \infty]$	$[2, 2]$	$[-\infty, \infty]$
3	$\bot$	$\bot$	$\bot$
$3'$	$\bot$	$\bot$	$\bot$
4	$\bot$	$\bot$	$\bot$
$4'$	$\bot$	$\bot$	$\bot$
5	$\bot$	$\bot$	$\bot$
6	$\bot$	$\bot$	$\bot$

$\rightsquigarrow$

	x	y	z
1	$[1, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[1, \infty]$	$[2, 2]$	$[-\infty, \infty]$
3	$[1, 2]$	$[2, 2]$	$[-\infty, \infty]$
$3'$	$\bot$	$\bot$	$\bot$
4	$[3, \infty]$	$[2, 2]$	$[-\infty, \infty]$
$4'$	$\bot$	$\bot$	$\bot$
5	$\bot$	$\bot$	$\bot$
6	$\bot$	$\bot$	$\bot$

$\rightsquigarrow$

	x	y	z
1	$[1, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[1, \infty]$	$[2, 2]$	$[-\infty, \infty]$
3	$[1, 2]$	$[2, 2]$	$[-\infty, \infty]$
$3'$	$[1, 2]$	$[2, 2]$	$[3, 6]$
4	$[3, \infty]$	$[2, 2]$	$[-\infty, \infty]$
$4'$	$[3, \infty]$	$[2, 2]$	$[2, 2]$
5	$\bot$	$\bot$	$\bot$
6	$\bot$	$\bot$	$\bot$

	x	y	z
1	$[1,\infty]$	$[-\infty,\infty]$	$[-\infty,\infty]$
2	$[1,\infty]$	$[2,2]$	$[-\infty,\infty]$
3	$[1,2]$	$[2,2]$	$[-\infty,\infty]$
$3'$	$[1,2]$	$[2,2]$	$[3,6]$
4	$[3,\infty]$	$[2,2]$	$[-\infty,\infty]$
$4'$	$[3,\infty]$	$[2,2]$	$[2,2]$
5	$[1,\infty]$	$[2,2]$	$[2,6]$
6	$\bot$	$\bot$	$\bot$

$\overset{*}{\rightsquigarrow}$ $\rightsquigarrow$

	x	y	z
1	$[1,\infty]$	$[-\infty,\infty]$	$[-\infty,\infty]$
2	$[1,\infty]$	$[2,2]$	$[-\infty,\infty]$
3	$[1,2]$	$[2,2]$	$[-\infty,\infty]$
$3'$	$[1,2]$	$[2,2]$	$[3,6]$
4	$[3,\infty]$	$[2,2]$	$[-\infty,\infty]$
$4'$	$[3,\infty]$	$[2,2]$	$[2,2]$
5	$[1,\infty]$	$[2,2]$	$[2,6]$
6	$[1,\infty]$	$[2,2]$	$[4,12]$

From this analysis we conclude that whenever $\mathsf{x} \in [1,\infty)$, z is at most 12 in line 6. In particular, this holds for all *odd* integers in $[1,\infty)$, which proves the property.

c) You can pick any trace which violates the program semantics, for example:

$$\langle 1, \{\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 0, \mathsf{z} \mapsto 0\}\rangle \to \langle 2, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0\}\rangle \to \cdots$$

Solution 3.

a) This is true. Let $x \in [a,b]$ and $y \in [c,d]$. Hence, $a \le x \le b$ and $c \le y \le d$. Now consider the sum $z := x+y$. It is $a+c \le z$ and $z \le b+d \le b+|d|$, because $d \le |d|$. Therefore, $z \in [a+c,\ b+|d|]$, which proves soundness.

b) This is not true. Consider the intervals $[0,0]$ and $[-1,-1]$, which only contain a single number each. Then, if $x \in [0,0]$ and $y \in [-1,-1]$, $x+y=-1$. However, the transformer gives $[0,0] +' [-1,-1] = [-1,1]$, which for example also includes 1.

c) This is not true. Consider $[-1,-1] +'' [0,0] = [-\infty,-2]$, which does not include value -1 produced by $-1+0$.

d) This is true. We need to prove that for any $z \in [a+5,\ b+5]$, there exists an integer x such that $z = x+5$ and $x \in [a,b]$. This can be shown by picking $x = z-5$: it is $a+5 \le x+5 \le b+5 \ \Leftrightarrow\ a \le x \le b$.

e) This is not true. Consider $[1,2] *' 2 = [2,4]$, which includes the number 3. However, the only possible results of multiplying an integer in $[1,2] = \{1,2\}$ by 2 are 2 and 4.

Note: Even though $*'$ is not precise, it is the best transformer in the interval domain for multiplication by the constant 2.

Exercise 10 - Solution

Dynamic Race Detection

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) $V_{critical} = \{\texttt{yes}, \texttt{no}\}$. We are looking for write-write and read-write conflicts. Note that `tmp` is re-declared for every usage, and is thus not shared between threads.

b) See Figure 1.

c) See Figure 1. The indices in the vector clocks are ordered according to `A`, `B-0`, `E-0`, `F-0`, `B-1`, `C-1`, `D-1`, `G`.

d) See Figure 1. The algorithm will not report a data race. Considering all pairs of nodes, we cannot find a pair that is both unordered and reads/writes from/to the same variable.

e) See Figure 2. This time, the algorithm detects a race: Nodes `E-0` and `F-1` are not ordered, yet both access the variable `yes`, the second access being a write.

f) Ordering the atomic actions as `A` $\rightarrow$ `B-0` $\rightarrow$ `B-1` $\rightarrow$ `E-0` $\rightarrow$ `E-1` $\rightarrow$ `F-0` $\rightarrow$ `F-1` $\rightarrow$ `G` prints "1 voted yes". However, ordering the atomic actions as `A` $\rightarrow$ `B-0` $\rightarrow$ `E-0` $\rightarrow$ `F-0` $\rightarrow$ `B-1` $\rightarrow$ `E-1` $\rightarrow$ `F-1` $\rightarrow$ `G` prints "2 voted yes".

g) We can model atomic threads by merging blocks `B-i, C-i, D-i, E-i, F-i` to a single Block `B'-i` (see Figure 3). In this case, nodes `B'-0` and `B'-1` both write to `yes`, yet they are unordered.

Note: The detected data race on variable `yes` leads to a non-deterministic read of `yes` in line 12: Depending on which thread is scheduled first, thread 0 either reads 0 or 1 for `yes`. However, this does not affect the print statements in lines 19–20 and the program overall behaves deterministically. This is an example of a "harmless race".

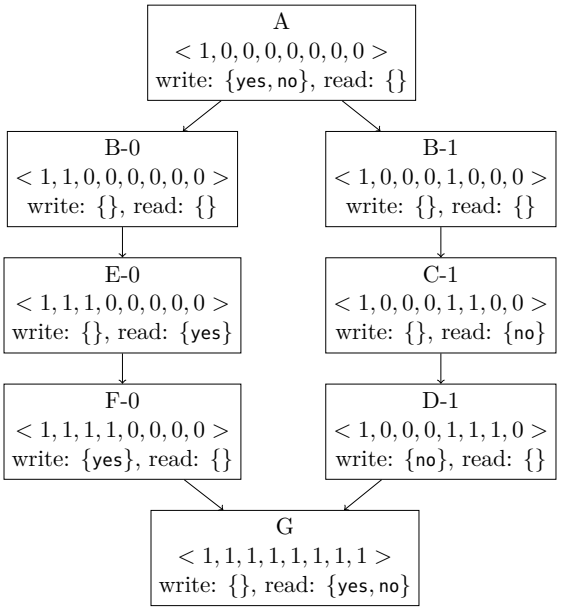


Figure 1: Happens-Before model for input `vote = [true, false]`.

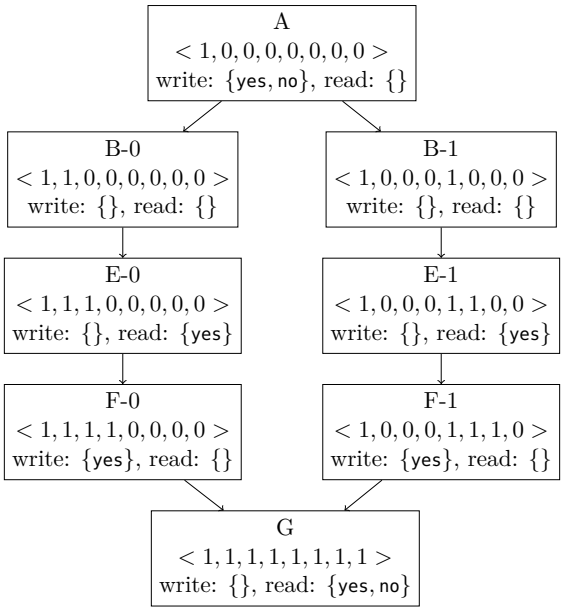


Figure 2: Happens-Before model for input `vote = [true, true]`.

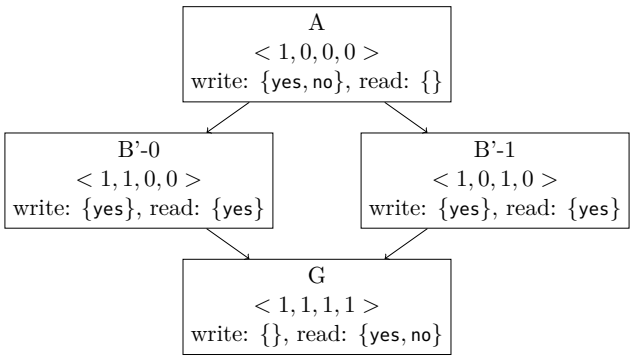


Figure 3: Happens-Before model for input `vote = [true, true]` using atomic threads.

Exercise 12 - Solution

Dynamic Alloy Models

Rigorous Software Engineering, ETH Zurich

Solution 1.

```
1  open util/integer
2  open util/ordering[Counter]
3
4  sig Counter {
5      n: Int
6  }
7
8  pred inc[c, c': Counter] {
9      c'.n = c.n.add[Int[1]]
10 }
11
12 pred init[c: Counter] {
13     c.n = 0
14 }
15
16 fact traces {
17     init[first] &&
18     all c: Counter - last | inc[c,c.next]
19 }
20
21 pred show { }
22
23 assert inv1 {
24     all c: Counter | 0 <= c.n
25 }
26
27 assert inv2 {
28     all c, c': Counter | lt[c, c'] => (c.n <= c'.n)
29 }
30
31 run show{} for 10 but 5 Int
32 check inv1 for 5
33 check inv2 for 5
```

Note: The `first`, `last` and `lt` functions are defined in the Alloy ordering library. [1]

[1] https://github.com/AlloyTools/org.alloytools.alloy/blob/master/org.alloytools.alloy.core/src/main/resources/models/util/ordering.als

Solution 2. We add the following predicate and fact to the model (see accompanying ZIP file for full solution).

```
open util/ordering[ETHBus]

pred drive[b1, b2: ETHBus] {
    one b1.station => no b2.station
    no b1.station => b2.station = b1.prev.station.next
}

fact route {
    first.station = Polyterrasse
    all b: ETHBus - last | drive[b, b.next]
}
```

Solution 3. The model defines a set of objects `Node` and a relation `next` $\subseteq$ `Node` $\times$ `Node`.

The given model has one constraint c: for every node n there is exactly one node m such that $(n, m) \in$ `next`. The assertion a checks whether for every node n there exists a node m with $(m, n) \in$ `next`.

Given two nodes n and m, we will use the boolean variable $x_{n,m}$ to denote $(n, m) \in$ `next`.

a) For the scope with one object, we have `Node` $= \{0\}$. We encode the constraint c as $x_{0,0}$, and the assertion a as $x_{0,0}$.

   The resulting boolean formula is $x_{0,0} \wedge \neg x_{0,0}$, which is not satisfiable. Therefore, for the given scope there is no counter-example for the assertion.

b) For the scope with two objects, we have `Node` $= \{0, 1\}$. We encode the constraint as

$$c := ((x_{0,0} \wedge \neg x_{0,1}) \vee (\neg x_{0,0} \wedge x_{0,1})) \wedge ((x_{1,0} \wedge \neg x_{1,1}) \vee (\neg x_{1,0} \wedge x_{1,1}))$$

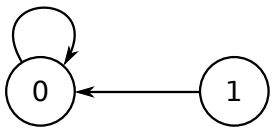
   and the assertion as $a := (x_{0,0} \vee x_{1,0}) \wedge (x_{0,1} \vee x_{1,1})$. The resulting boolean formula is

$$c \wedge \neg a.$$

   This formula is satisfied for

$$x_{0,0} = T \quad x_{0,1} = F \quad x_{1,0} = T \quad x_{1,1} = F,$$

   which corresponds to the following counter-example:



c) The new field and fact result in two additional constraints: (c_1) every node has exactly one previous node, and (c_2) for every node n, there exists a node m such that $(n, m) \in$ `next` and $(m, n) \in$ `prev`. We write $p_{n,m}$ to denote $(n, m) \in$ `prev`.

Scope 1: For checking `check demo for 1`, we encode the constraints as:

$$\begin{aligned} x_{0,0} \quad & \text{(Constraint } c\text{)} \\ p_{0,0} \quad & \text{(Constraint } c_1\text{)} \\ x_{0,0} \wedge p_{0,0} \quad & \text{(Constraint } c_2\text{)} \end{aligned}$$

and the assertion is encoded as before:

$$x_{0,0} \qquad \text{(Assertion } a\text{)}.$$

The resulting boolean formula is

$$(x_{0,0} \wedge p_{0,0} \wedge (x_{0,0} \wedge p_{0,0})) \wedge \neg x_{0,0}.$$

This formula is not satisfiable, because the conjunction contains both $x_{0,0}$ and $\neg x_{0,0}$. Therefore, there is no counter-example for the given scope.

Scope 2: For checking `check demo for 2`, we encode the constraints as:

$$\begin{aligned} c &:= ((x_{0,0} \wedge \neg x_{0,1}) \vee (\neg x_{0,0} \wedge x_{0,1})) \wedge ((x_{1,0} \wedge \neg x_{1,1}) \vee (\neg x_{1,0} \wedge x_{1,1})) \\ c_1 &:= ((p_{0,0} \wedge \neg p_{0,1}) \vee (\neg p_{0,0} \wedge p_{0,1})) \wedge ((p_{1,0} \wedge \neg p_{1,1}) \vee (\neg p_{1,0} \wedge p_{1,1})) \\ c_2 &:= ((x_{0,0} \wedge p_{0,0}) \vee (x_{0,1} \wedge p_{1,0})) \wedge ((x_{1,0} \wedge p_{0,1}) \vee (x_{1,1} \wedge p_{1,1})) \end{aligned}$$

and the assertion is encoded as before:

$$a := (x_{0,0} \vee x_{1,0}) \wedge (x_{0,1} \vee x_{1,1}).$$

The resulting boolean formula is

$$c \wedge c_1 \wedge c_2 \wedge \neg a.$$

As can be checked by a SAT solver, this formula is not satisfiable. Therefore, there is no counter-example for the given scope.

Exercise 03 - Solution

Structural Testing

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) We only need 9 test cases as shown below.
Hint: It is usually easiest to start with the parameters that have the most possible values.

name	val	limit	optimize
Alice	0	0	**true**
Alice	1	10	**false**
Alice	2	20	**false**
Bob	0	10	**false**
Bob	1	20	**true**
Bob	2	0	**true**
Charlie	0	20	**true**
Charlie	1	0	**false**
Charlie	2	10	**true**

b) (i) This is true. There are $a \cdot b$ possible value combinations for the two parameters with largest domain. Each of these combinations needs to occur in a separate test case, hence we need at least $a \cdot b$ test cases.

(ii) This is not true. In particular, consider the case of four booleans. It is not possible to cover all pairwise value combinations using only $2 \cdot 2 = 4$ test cases (however, you have seen in the lecture that it is possible using 5 test cases). Below, we show a quite advanced proof.

Proof. Assume for the sake of contradiction that there exists a set $\mathcal{T}$ of four test cases covering all pairwise value combinations of four booleans. We can assemble these test cases in a table Z similarly as in subtask (a). The table Z has 4 columns and 4 rows.

Observation 1: Each of the columns of Z must consist of exactly two entries F (false) and two entries T (true). Otherwise, Z would not cover all value pairs (T, T), (T, F), (F, T), (F, F) between any two columns.

Observation 2: Negating a whole column of Z will result in a table which (i) still represents valid pairwise testing inputs, and (ii) still satisfies observation 1.

Now, in Z, we negate all columns where the top entry is F to obtain the table Z'. The top row of Z' hence only consists of T entries. Due to observation 2, Z' also represents valid pairwise testing inputs. However, due to observation 1, the only possible columns that may appear in Z' are:

T	T	T
T	F	F
F	T	F
F	F	T

Because there are 4 columns in the Z', at least two columns must have identical values. For these columns, only the value pairs (T, T) and (F, F) are covered. Hence, the test set represented Z' is invalid—a contradiction. □

Solution 2.

a) This is true. Assume T has 100% branch coverage in P. Using structural induction over the control flow graph (CFG) of P, we can prove that T reaches every statement in P. Note that the CFG of P is acyclic, as P is loop-free.

Induction hypothesis: For a node (basic block) n in the CFG, let $H(n)$ denote the fact that some $t \in T$ reaches n.

Base case: All inputs in T trivially reach the "entry" node (for this, we need T to be non-empty). Hence, $H(\text{entry})$ holds.

Step case: Assume $H(n)$ for some basic block n in the CFG, and let $T_n \subseteq T$ be the non-empty set of test inputs reaching n. We can prove $H(m)$ for all successors m of n in the CFG as follows. If m is the only successor, all $t \in T_n$ are guaranteed to reach m. If n has multiple successors, 100% branch coverage guarantees that there exists at least one $t \in T_n$ that reaches m.

By induction, H holds for all basic blocks of P. Hence, every basic block (and therefore every statement) of P is reached by at least one test input $t \in T$.

b) This is not true. Consider the simple program below, which has 2 program paths. The test input set {true} has path coverage 50%, but only statement coverage $2/5 = 40\%$.

```
void main(boolean a) {
  if (a) {
    a = !a;
  } else {
    a = !a;
    a = !a;
    a = !a;
  }
}
```

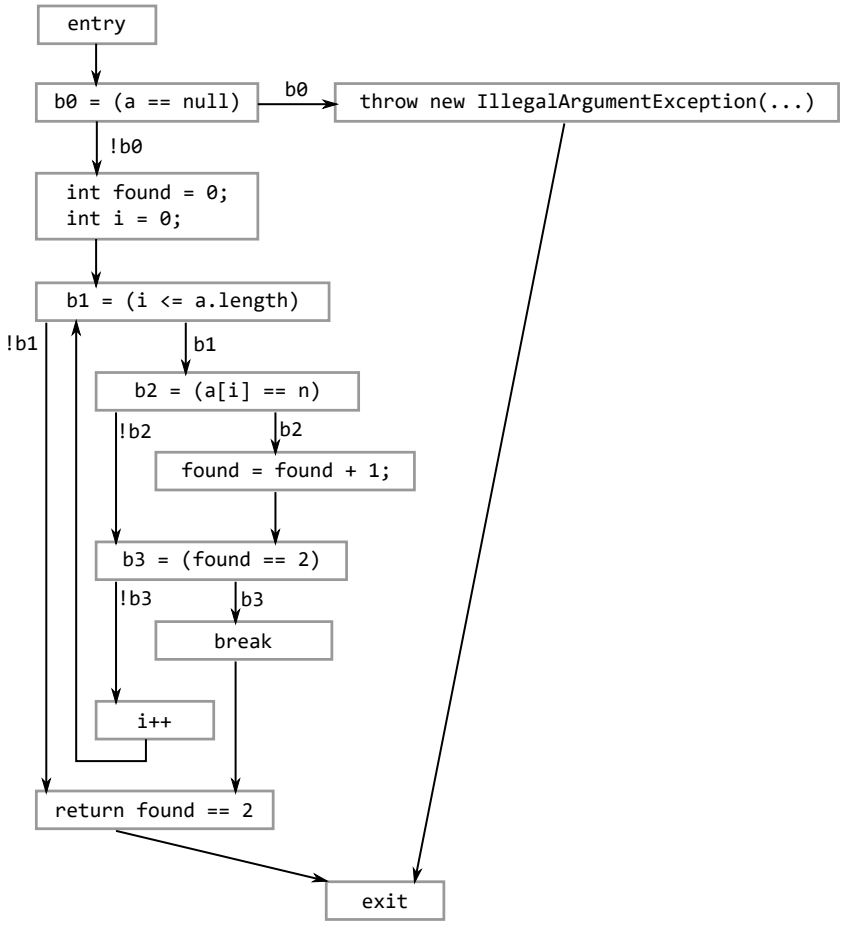


Figure 1: Control flow graph of `hasDuplicate`.

Solution 3.

a) See Fig. 1.

b) Method `hasDuplicate`:

(i) The implementation throws an out-of-bounds array access exception for inputs where `n` does not occur in `a`. For example:

$$\texttt{a} = [1, 2], \quad \texttt{n} = 3.$$

(ii) The following tests achieve 100% statement coverage but miss the bug.

$$\texttt{a} = \textbf{null}, \quad \texttt{n} = 0$$
$$\texttt{a} = [1, 1], \quad \texttt{n} = 1$$

(iii) No. At least one test input in the set would have to take branch `!b1`, which is

only possible if the loop is not exited early by the **`break`** statement. However, this will always find the out-of-bounds access in the last loop iteration.

(iv) No, it is not even possible to achieve 100% loop coverage here. For the loop to execute 0 times, we would need to reach the node `b1 = (i <= a.length)` and take branch `!b1` when `i` is still zero. Because `a.length` is always non-negative, this cannot happen.

Method `average`:

(i) The implementation skips the first element of the array and returns an incorrect result if the first element is non-zero. For example:

$$\texttt{a} = [1].$$

(ii) The following tests achieve 100% statement coverage but miss the bug.

$$\texttt{a} = \textbf{\texttt{null}}$$
$$\texttt{a} = []$$
$$\texttt{a} = [0, 1]$$

(iii) The tests from (ii) achieve 100% branch coverage but miss the bug.

(iv) The following tests achieve 100% loop coverage but miss the bug.

$$\texttt{a} = [0]$$
$$\texttt{a} = [0, 1]$$
$$\texttt{a} = [0, 1, 2]$$

Method `averageSubarray`:

(i) The implementation throws a division-by-zero exception if `a` is not **`null`** and `l` = `r`. For example:

$$\texttt{a} = [1, 2], \quad \texttt{l} = 1, \quad \texttt{r} = 1.$$

(ii) The following tests achieve 100% statement coverage but miss the bug.

$$\texttt{a} = \textbf{\texttt{null}}, \quad \texttt{l} = 1, \quad \texttt{r} = 1$$
$$\texttt{a} = [1, 2], \quad \texttt{l} = 0, \quad \texttt{r} = 2$$

(iii) The tests from (ii) achieve 100% branch coverage but miss the bug.

(iv) No. Executing the **`while`**-loop zero times will reveal the division-by-zero.

Exercise 08 - Solution

Verifying Determinism

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) In each of the iterations below, we apply all assignments in the order they appear in the program. However, note that the order of application does not matter and there are multiple ways to obtain the result. We write aX to denote the allocation site at line X.

k1	⊥
k2	⊥
n0	⊥
n	⊥
c	⊥
x	⊥

$\rightsquigarrow$

k1	{ a1 }
k2	{ a2 }
n0	{ a3 }
n	{ a3, a8, a12 }
c	{ a8, a12 }
x	{ null, a8, a12 }
a1.p	{ null }
a1.next	{ null }
a2.p	{ null }
a2.next	{ null }
a3.p	{ null, a1 }
a3.next	{ null, a8, a12 }
a8.p	{ null, a1, a2 }
a8.next	{ null, a8, a12 }
a12.p	{ null, a1 }
a12.next	{ null }

$\rightsquigarrow$

k1	{ a1 }
k2	{ a2 }
n0	{ a3 }
n	{ a3, a8, a12 }
c	{ a8, a12 }
x	{ null, a8, a12 }
a1.p	{ null }
a1.next	{ null }
a2.p	{ null }
a2.next	{ null }
a3.p	{ null, a1 }
a3.next	{ null, a8, a12 }
a8.p	{ null, a1, a2 }
a8.next	{ null, a8, a12 }
a12.p	{ null, a1, a2 }
a12.next	{ null, a8, a12 }

We can prove A2 using this result, since { null, a8, a12 } $\cap$ { a3 } $= \emptyset$.

b) Note that the non-deterministic condition $*$ does not allow us to learn anything about the state after entering or leaving the loop. Hence, when processing line 7, we simply propagate the state to lines 8 and 17.

The fixed-point state at line 19 is given below (intermediate iterations and other labels not shown).

k1	{ a1 }
k2	{ a2 }
n0	{ a3 }
n	{ a3, a12 }
c	$\top$
x	{ null, a8 }
a1.p	{ null }
a1.next	{ null }
a2.p	{ null }
a2.next	{ null }
a3.p	{ null, a1 }
a3.next	{ null, a8 }
a8.p	{ null, a2 }
a8.next	{ null, a12 }
a12.p	{ null, a1 }
a12.next	{ null, a8 }

Again, we can only prove A2 using this result. For A3, note that {null, a2} $\cap$ {null, a1} = {null} and hence both `x.p` and `x.next.p` could be null, which violates the assertion.

c) Note how the **`while`** loop builds up an acyclic list connected by `next` pointers, alternatingly attaching `k1` or `k2` to the list elements via `p` pointers. Starting from `n0`, it appends new objects created in lines 8 and 12, and moves the `n` pointer forward along the list.

Assuming the pointer dereferences in A1–A3 are valid, all these assertions hold.

Solution 2.

a) The only pointer assignments in the program are on lines 7 and 9. Additionally, we have to consider the allocation site A for the input. Performing flow-insensitive pointer analysis gives the following points-to sets.

$$\texttt{input} \mapsto \{A\}$$
$$\texttt{output} \mapsto \{A, B\}$$

b) The fixed-point for the fork contexts is shown below.

	n	half	i	j
16	$[10, 10]$	$[5, 5]$	$\bot$	$\bot$
17	$[10, 10]$	$[5, 5]$	$\bot$	$\bot$
18	$[10, 10]$	$[5, 5]$	$[1, 4]$	$\bot$
19	$[10, 10]$	$[5, 5]$	$[1, 4]$	$\bot$
20	$[10, 10]$	$[5, 5]$	$[1, 4]$	$\bot$
21	$[10, 10]$	$[5, 5]$	$[5, 5]$	$\bot$

	n	half	i	j
23	$[10, 10]$	$[5, 5]$	$\bot$	$\bot$
24	$[10, 10]$	$[5, 5]$	$\bot$	$\bot$
25	$[10, 10]$	$[5, 5]$	$\bot$	$[6, 9]$
26	$[10, 10]$	$[5, 5]$	$\bot$	$[6, 9]$
27	$[10, 10]$	$[5, 5]$	$\bot$	$[6, 9]$
28	$[10, 10]$	$[5, 5]$	$\bot$	$[10, 10]$

c) First, note that the two groups of lines 16–21 and 23–28 are executed by a single thread each and we hence only have to check conflicts for pairs of lines across these two groups. According to the pointer analysis from (a), `input` may alias with `output`. The only pairs of lines where a write-write or a read-write conflict may occur are: $(16, 23), (16, 25), (16, 26), (18, 23), (18, 26), (19, 23), (19, 25)$, and $(19, 26)$. For each of these pairs, it can be shown that no data-races occur, using the result from (b):

- $(16, 23)$: The indices used to read from and write to the arrays are disjoint: $[0, 0] \cap [5, 5] = \emptyset$.
- $(18, 26)$: Line 18 performs two reads at indices $[0, 3]$ and $[1, 4]$. Line 26 writes to indices $[6, 9]$. No data-race is possible here, as $([0, 3] \cup [1, 4]) \cap [6, 9] = \emptyset$.
- etc. (similar arguments for other pairs)

Exercise 09 - Solution

Symbolic Execution

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) See the code below. Note that the if-statements generally need to be nested as evaluating the condition may have side-effects (no problem in this case, however).

```
     int bounded_gcd(int a, int b) {
1:       if (b != 0) {
2:           // first iteration
3:           int tmp = b;
4:           b = a mod b;
5:           a = tmp;
6:
7:           if (b != 0) {
8:               // second iteration
9:               int tmp = b;
10:              b = a mod b;
11:              a = tmp;
12:          }
13:      }
14:      return a;
     }
```

b) We start with the initial symbolic store $\sigma_s = \{\texttt{a} \mapsto a_0, \texttt{b} \mapsto b_0\}$ and the path constraint $\text{pct} : a_0 > 0 \wedge b_0 > 0$ modelling positive parameters. Then, we perform symbolic execution in a depth-first manner as shown below.

(i) Line 1: We consider both entering and not entering the body of the if-statement, see (ii) and (viii).

(ii) Line 2: From the condition `b != 0` we construct the constraint $b_0 \neq 0$ and add it to the path constraint, which can be simplified. The store is unchanged.

$$\sigma_s = \{\texttt{a} \mapsto a_0, \texttt{b} \mapsto b_0\}$$
$$\text{pct} : a_0 > 0 \wedge b_0 > 0 \wedge b_0 \neq 0 \;\equiv\; a_0 > 0 \wedge b_0 > 0$$

The new path constraint is satisfiable by $a_0 = b_0 = 1$. We continue with (iii).

(iii) Lines 3–6: The symbolic store is updated according to the assignments, see below. We continue with (iv).

$$\sigma_s = \{\texttt{a} \mapsto b_0,\ \texttt{b} \mapsto a_0 \,\%\, b_0,\ \texttt{tmp} \mapsto b_0\}$$
$$\text{pct} : a_0 > 0 \wedge b_0 > 0$$

(iv) Line 7: We can either enter or not enter the body, see (v) and (vii).

(v) Line 8: We add $a_0 \,\%\, b_0 \neq 0$ to the path constraint as shown below.

$$\sigma_s = \{\texttt{a} \mapsto b_0,\ \texttt{b} \mapsto a_0 \,\%\, b_0,\ \texttt{tmp} \mapsto b_0\}$$
$$\text{pct} : a_0 > 0 \wedge b_0 > 0 \wedge a_0 \,\%\, b_0 \neq 0$$

The constraint is still satisfiable (e.g., $a_0 = 1, b_0 = 2$). We continue with (vi).

(vi) Lines 9–14: The symbolic store is updated according to the assignments. We continue up to line 14, where the function returns. The final symbolic state is given below.

$$\sigma_s = \{\texttt{a} \mapsto a_0 \,\%\, b_0,\ \texttt{b} \mapsto b_0 \,\%\, (a_0 \,\%\, b_0)\}$$
$$\text{pct} : a_0 > 0 \wedge b_0 > 0 \wedge a_0 \,\%\, b_0 \neq 0$$

(vii) Lines 13-14: We update the path constraint as shown below.

$$\text{pct} : a_0 > 0 \wedge b_0 > 0 \wedge a_0 \,\%\, b_0 = 0$$

The constraint is still satisfiable (e.g., $a_0 = 1, b_0 = 1$). We continue up to line 14, where the function returns. The final symbolic state is given below.

$$\sigma_s = \{\texttt{a} \mapsto b_0,\ \texttt{b} \mapsto a_0 \,\%\, b_0\}$$
$$\text{pct} : a_0 > 0 \wedge b_0 > 0 \wedge a_0 \,\%\, b_0 = 0$$

(viii) Line 14: We add the constraint $b_0 = 0$ to the path constraint and obtain $\text{pct} : a_0 > 0 \wedge b_0 > 0 \wedge b_0 = 0$. This constraint is not satisfiable, so we stop exploration.

In summary, the symbolic states at the return statement are given in (vi) and (vii).

c) For example:

$$\texttt{a} = 10$$
$$\texttt{b} = 15$$
$$\texttt{gcd(a, b)} = 5$$
$$\texttt{bounded\_gcd(a, b)} = 10$$

The symbolic state generated by symbolic execution on `bounded_gcd` includes *more* values than can be returned by `gcd`. For example, the state (vi) from subtask (b) includes the output 10 from the concrete example above. This means that symbolic execution on `bounded_gcd` is not an under-approximation of `gcd` any more.

d) The main idea is to suppress traces which would need a third iteration of the loop. A simple way to achieve this is to introduce a special **impossible** statement, which changes the path constraint to **false** when reached. Loop unrolling would then work as shown below.

```
int new_bounded_gcd(int a, int b) {
    if (b != 0) {
        // first iteration
        int tmp = b;
        b = a mod b;
        a = tmp;

        if (b != 0) {
            // second iteration
            int tmp = b;
            b = a mod b;
            a = tmp;

            if (b != 0) {
                // suppress third iteration
                impossible;
            }
        }
    }
    return a;
}
```

Symbolic execution will now only produce a satisfiable path constraint for `b` becoming zero after 0, 1, or 2 iterations. If it is still non-zero after 2 iterations, the **impossible** statement stops symbolic execution. To see how this resolves the issue from (c), you can apply symbolic execution to `new_bounded_gcd`.

Solution 2.

a) We execute `main` using concrete inputs $\texttt{b} = \texttt{e} = 0$ as normal, but build a symbolic store and path constraint for the specific control flow taken under these inputs. The initial symbolic store and path constraint are $\sigma_s = \{\texttt{b} \mapsto b_0, \texttt{e} \mapsto e_0\}$ and pct : true. The gathered path constraint consists of 3 parts.

(i) In `pow`, we don't enter the loop body since the loop condition `i < e` evaluates to false. This adds $1 \geq e_0$ to the path constraint.

(ii) We enter the body of the first if-statement in `main`. This adds $e_0 \ \% \ 2 = 0$ to the path constraint.

(iii) We don't enter the body of the nested if-statement in `main`, which adds $b_0 \geq 0$ to the path constraint.

The gathered path constraint is hence pct : $1 \geq e_0 \ \wedge \ e_0 \ \% \ 2 = 0 \ \wedge \ b_0 \geq 0$.

b) We negate the sub-constraint from (iii) to obtain the following path constraint.

$$\text{pct}: \quad 1 \geq e_0 \ \wedge \ e_0 \ \% \ 2 = 0 \ \wedge \ b_0 < 0$$

A satisfying assignment is for example $e_0 = 0,\ b_0 = -1$.

c) In this run, we enter the body of the nested if-condition and reach the assertion. The gathered path constraint is the constraint from (b).

Note how we were automatically able to find a test input ($\texttt{e} = 0,\ \texttt{b} = -1$) violating the assertion! You may now find the bug in `pow` and fix it.

d) In this case, `pow` neither extends the path constraint, nor returns a symbolic value. Instead, we concretely execute `pow(0,0)` to obtain the value 0 for `r`. After the first line of `main`, the symbolic store is concretized: $\sigma_s = \{\texttt{b} \mapsto 0, \texttt{e} \mapsto 0, \texttt{r} \mapsto 0\}$.

Note: In addition to the return value, we also need to concretize the function arguments of `pow` in the symbolic state for the result to be an under-approximation of the concrete state. In particular, if b_0 was not concretized, the symbolic store and path constraint after the first line of `main` would admit the satisfying assignment $\texttt{b} \mapsto 1, \texttt{e} \mapsto 0, \texttt{r} \mapsto 0$, which cannot result from a concrete execution.

Like in (a), we enter the body of the first if-condition in `main`, and don't enter the body of the nested if-condition. However, because the variables are already concretized, we don't extend the path constraint for these conditions.

The resulting path constraint is simply pct : true.

e) No, we cannot proceed analogously as in subtasks (b–c). Because `r` is concretized, there is no sub-constraint we could negate in order to enter the second if-condition. The vanilla concolic execution presented here is not going to reach the assertion, unless it is lucky enough to select a "bad" concrete input right from the beginning.

f) An analogous C implementation is given below.

```
int my_pow(int b, int e) {
    int r = b;
    for (int i = 1; i < e; i++) {
        r = r * b;
    }
    return r;
}

void main(int b, int e) {
    int r = my_pow(b, e);
    if (e % 2 == 0) {
        if (r < 0) {
            pathcrawler_assert(0);
        }
    }
}
```

Exercise 04 - Solution

First Steps with Abstract Interpretation

Rigorous Software Engineering, ETH Zurich

Consider the following program P. Based on manual inspection, we believe that the value of x at the end of P is always even and negative, and line 3 is unreachable (see the three assertions below). In this exercise, we are trying to prove these properties.

```
   function(int x, int y) {
0:    x := y * 2
1:    while x >= 0
2:       if x = 1
3:          x := x + 1   // assert not reachable
          else
4:          y := x - 1
5:          x := y - 1
6:    skip               // assert x even
                         // assert x < 0
   }
```

Solution 1.

a) Trace for `function(5, 0)`:

$$\langle 0, \{\mathsf{x} \mapsto 5, \mathsf{y} \mapsto 0\}\rangle \to \langle 1, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 0\}\rangle \to \langle 2, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 0\}\rangle \to \langle 4, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 0\}\rangle$$
$$\to \langle 5, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto -1\}\rangle \to \langle 1, \{\mathsf{x} \mapsto -2, \mathsf{y} \mapsto -1\}\rangle \to \langle 6, \{\mathsf{x} \mapsto -2, \mathsf{y} \mapsto -1\}\rangle$$

Trace for `function(3, 1)`:

$$\langle 0, \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 1\}\rangle \to \langle 1, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1\}\rangle \to \langle 2, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1\}\rangle \to \langle 4, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1\}\rangle$$
$$\to \langle 5, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1\}\rangle \to \langle 1, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 1\}\rangle \to \langle 2, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 1\}\rangle \to \langle 4, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 1\}\rangle$$
$$\to \langle 5, \{\mathsf{x} \mapsto 0, \mathsf{y} \mapsto -1\}\rangle \to \langle 1, \{\mathsf{x} \mapsto -2, \mathsf{y} \mapsto -1\}\rangle \to \langle 6, \{\mathsf{x} \mapsto -2, \mathsf{y} \mapsto -1\}\rangle$$

b) No states with label 3 occur in the traces. The traces hence satisfy the non-reachability assertion. For label 6, only the state $\langle 6, \{\mathsf{x} \mapsto -2, \mathsf{y} \mapsto -1\}\rangle$ occurs in the traces, which satisfies both the evenness and the negativity assertion.

c) To prove an assertion, we have to show that it holds for any concrete state in $[\![P]\!]$ at the respective label. If $[\![P]\!]$ is finite, we could use it to prove the assertions. For instance, to verify non-reachability, we could check whether this set contains any states with label 3.

However, $[\![P]\!]$ may be infinitely large. In this case, it's not clear how we could use it to e.g. prove evenness without giving up termination: simply looping through the infinitely many elements of $[\![P]\!]$ may not terminate.

d) Due to Rice's theorem, constructing such a membership check ($s \overset{?}{\in} [\![P]\!]$) is not possible in general. Otherwise, we could use this to solve the undecidable halting problem by asking the algorithm whether the final label occurs in $[\![P]\!]$.

Solution 2.

a) We should define even `+ 1` $= \mathit{odd}$, because increasing any even number by 1 results in an odd number. Further, we should define $\top$ `* 2` $= \mathit{even}$ because any integer multiplied by 2 results in an even number.

b) The iterations are as follows:

pc	x	y
0	$\top$	$\top$
1	$\bot$	$\bot$
2	$\bot$	$\bot$
3	$\bot$	$\bot$
4	$\bot$	$\bot$
5	$\bot$	$\bot$
6	$\bot$	$\bot$

$\rightsquigarrow$

pc	x	y
0	$\top$	$\top$
1	*even*	$\top$
2	$\bot$	$\bot$
3	$\bot$	$\bot$
4	$\bot$	$\bot$
5	$\bot$	$\bot$
6	$\bot$	$\bot$

$\rightsquigarrow$

pc	x	y
0	$\top$	$\top$
1	*even*	$\top$
2	*even*	$\top$
3	$\bot$	$\bot$
4	$\bot$	$\bot$
5	$\bot$	$\bot$
6	*even*	$\top$

$\rightsquigarrow$

pc	x	y
0	$\top$	$\top$
1	*even*	$\top$
2	*even*	$\top$
3	$\bot$	$\bot$
4	*even*	$\top$
5	$\bot$	$\bot$
6	*even*	$\top$

$\rightsquigarrow$

pc	x	y
0	$\top$	$\top$
1	*even*	$\top$
2	*even*	$\top$
3	$\bot$	$\bot$
4	*even*	$\top$
5	*even*	*odd*
6	*even*	$\top$

$\rightsquigarrow$

pc	x	y
0	$\top$	$\top$
1	*even*	$\top$
2	*even*	$\top$
3	$\bot$	$\bot$
4	*even*	$\top$
5	*even*	*odd*
6	*even*	$\top$

(fixed-point!)

Notes: In the third iteration, the condition `x = 1` is guaranteed to be false because `x` is even, that's why we don't propagate to line 3. Even though nothing changes in the last iteration, it involves a join operation at line 1.

c) Yes, the state $\langle 6, \{\mathsf{x} \mapsto -2, \mathsf{y} \mapsto -1\}\rangle$ is covered by $\{\mathsf{x} \mapsto \mathit{even}, \mathsf{y} \mapsto \top\}$ at pc 6. This is ensured by the soundness property of abstract interpretation.

d) The fixed-point state assigns $\bot$ to both `x` and `y` at pc 3. This means that this line is never reached, and the non-reachability assertion is guaranteed to hold.

At pc 6, the fixed-point state assigns *even* to `x`. Hence, `x` is guaranteed to be even in line 6 and the assertion holds. However, we can not conclude that $\texttt{x} < 0$, even though this is actually true (why?). The *Parity* domain is too imprecise for this.

Solution 3. While the domains Sign and Interval are comparable, Parity is not comparable with both Sign and Interval.

- The Interval domain is more precise than Sign because any state expressed with the Sign domain can be exactly expressed with the Interval domain (e.g., $+ \equiv [0, \infty)$). The converse does not hold, there are states expressed using the Interval domain that cannot be exactly expressed using the Sign domain (e.g., $[1, 2]$).
- The following program can be verified using the Interval and Sign domain, but cannot be verified with the Parity domain:

```
int x := 1;
assert x >= 0;
```

The following program can be verified using the Parity domain, but cannot be verified with the Interval or Sign domain:

```
foo(int i) {
    x := 2*i;
    assert x is even;
}
```

Exercise 02 - Solution

Modularity and Functional Testing

Rigorous Software Engineering, ETH Zurich

Solution 1. The examples are mostly taken from an answer of user "BalusC" to a StackOveflow question. [1]

a) Observer. This is a listener class whose method `windowClosing` serves as "update" function called whenever a window is closed.

b) Factory or Flyweight. The method `getInstance()` can be viewed as a factory method creating objects of type `NumberFormat`. However, because the method is **`static`**, it does not completely follow the the (abstract) factory pattern we have seen in the lecture, where an actual factory object is used. Alternatively, the returned `NumberFormat` objects can be considered flyweights, where the overloaded method with one parameter allows to provide the key for the requested flyweight.

c) Visitor. See all the different `visit` functions.

d) Observer. This event is passed by an `SSLSession` to listener objects whenever a value is put to or removed from the session.

e) Strategy. The `compare` method can be implemented in various ways to achieve different, specialized behavior.

f) Factory. The method `newDocumentBuilder()` is a standard factory method creating `DocumentBuilder` objects based on the configuration of the factory.

g) Flyweight. This method creates immutable `Boolean` flyweights.

h) Strategy. The `doFilter` method can be implemented in various ways to achieve different, specialized behavior.

i) Visitor. The `accept` method applies a `TypeVisitor`.

j) Flyweight. This method creates immutable `Character` flyweights.

[1] `http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns`

Solution 2. This task is inspired by Wikipedia's example of the abstract factory pattern. [2]

a) Currently, clients are tightly coupled to the OS-specific button implementations. In particular, clients need to know and choose the actual button class matching the current OS. This creates issues with maintainability, for instance if the framework's developers decide to add support for Linux operating systems, change the way buttons are created in different OSs, or switch to a more powerful button implementation supporting multiple OSes. In a way, the different button classes are just implementation details.

b) We can mitigate the issue using an abstract factory. In particular, we introduce button factories as shown below:

```
public interface ButtonFactory {
    public Button create();
}

public class WinFactory implements ButtonFactory {
    @Override
    public Button create() {
        return new WinButton();
    }
}

public class OSXFactory implements ButtonFactory {
    @Override
    public Button create() {
        return new OSXButton();
    }
}
```

Then, the GUI framework can provide concrete factories to clients:

```
public class GUIFramework {
    public static ButtonFactory getButtonFactory(String os) {
        if (os.equals("win")) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
}
```

Finally, a client just has to request a button factory for a specific operating system, without having to worry about the different implementations of `Button`. For example:

```
ButtonFactory factory = GUIFramework.getButtonFactory("win");
Button button = factory.create();
button.display();
```

[2] https://en.wikipedia.org/wiki/Abstract_factory_pattern

Solution 3.

a) The classes exhibit the following main issues:

- `StudentsManager` exposes its internal representation by making `nStudents` and `studentIds` public. This allows the client code to access `studentIds` directly. If we were to refactor the internal representation of `StudentsManager`, the client code may no longer compile.
- The constructor of `StudentsManager` captures the pointer to `studentIds` (i.e., it is stored without cloning first). If the client maintains the pointer to the set, it can directly manipulate the internal representation of `StudentsManager` (e.g., remove students and thereby invalidate `nStudents`).
- `GradesManager` inherits from `StudentsManager` for no good reason. First, we don't need a subtype relation between the two classes. Also, `GradesManager` (possibly accidentally) inherits the method `addStudent`: calling this method on `GradesManager` would not result in the intended behavior. Finally, `GradesManager` reads the field `nStudents`, which is part of the internal representation of the class `StudentsManager` (note that even if `nStudents` was private, this would still compile). Consider the following refactoring: Remove `nStudents` from `StudentsManager` and replace the body of `getNumberOfStudents` by the statement `return this.studentIds.size();`. In this case, `GradesManager` will no longer compile (*fragile baseclass problem*).
- *(non-severe issue)* `GradesManager` is coupled to `HashSet` due to the allocation in the constructor. We would have to change the code of `GradesManager` if we e.g. want to use a more efficient set implementation.

b) A possible refactoring addressing the above issues is shown below. The two classes are still coupled to `HashSet` via allocations. To further reduce coupling, we could additionally use an abstract factory for the sets (see also the previous task).

```
public class StudentsManager {
    private int nStudents;              // information hiding
    private Set<Integer> studentIds;    // information hiding

    public StudentsManager(Set<Integer> studentIds) {
        this.studentIds = new HashSet<Integer>(studentIds);   // no capturing
        this.nStudents = studentIds.size();
    }

    public void addStudent(int id) {
        if (!this.studentIds.contains(id)) {
            this.nStudents++;
            this.studentIds.add(id);
        }
    }

    public int getNumberOfStudents() {
```

```
        return this.nStudents;
    }


    public GradesManager prepareGrading() {     // more natural for client
        return GradesManager(this.studentIds);
    }
}

public class GradesManager {    // no inheritance
    private int nStudents;      // no coupling to the internal
                                // representation of StudentsManager
    private double rollingSum = 0.0;
    private Set<Integer> studentsWithoutGrades;

    public GradesManager(Set<Integer> studentIds) {
        this.studentsWithoutGrades = new HashSet<Integer>(studentIds);
        this.nStudents = studentIds.count();
    }

    public void setGrade(int id, double grade) {
        if (this.studentsWithoutGrades.contains(id)) {
            this.rollingSum += grade;
            this.studentsWithoutGrades.remove(id);
        }
    }

    public double computeAverageGrade() {
        if (!this.studentsWithoutGrades.isEmpty())
            throw new RuntimeException("Not every student is assigned a grade!");
        return this.rollingSum / this.nStudents;
    }
}
```

Client code:

```
StudentsManager sMgr = new StudentsManager(new HashSet<Integer>());
sMgr.addStudent(10); sMgr.addStudent(20);

GradesManager gMgr = sMgr.prepareGrading();     // no representation exposure
gMgr.setGrade(10, 4.5); gMgr.setGrade(20, 6.0);
gMgr.computeAverageGrade();
```

Solution 4.

a) A good choice for equivalence classes is the following:

- Empty input array. In this case, the median is 0. *Representative input: []*
- Non-empty input array with odd length. In this case, the median is the middle number in the ordered array. *Representative input: [1.6, 0.3, 4.8]*

- Non-empty input array with even length. In this case, the median is the arithmetic mean of the two middle numbers in the ordered array. *Representative input: [3.0, 2.0, 4.5, 1.3]*

b) See `MedianPartitionsTest.java` in the accompanying solution ZIP file. Line 12 in the original `Median.java` contains an error, which is fixed in the solution.

c) See `MedianParameterizedTest.java`.

d) See `MedianOracleTest.java`.

e) See `MedianRandomizedTest.java`.

Exercise 05 - Solution

Mathematical Concepts

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) Assume (for the sake of contradiction) there exist two distinct least elements $\bot$ and $\bot'$ such that $\bot \neq \bot'$. By definition, it must be $\forall p \in L.\ \bot \sqsubseteq p$, in particular $\bot \sqsubseteq \bot'$. By a symmetric argument, it must be $\bot' \sqsubseteq \bot$. Due to anti-symmetry of $\sqsubseteq$, it follows that $\bot = \bot'$, which is a contradiction.

b) (i) This is true. The three properties required for $\preceq$ can be shown as follows.

Reflexivity: It is $a \leq a$ and $(a - a) \bmod 3 = 0$ for all $a \in \mathbb{N}$, and hence $a \preceq a$.

Transitivity: Assume $a \preceq b$ and $b \preceq c$ for some $a, b, c \in \mathbb{N}$. It is $a \leq b$ and $b \leq c$, hence $a \leq c$. Also, it is $(b-a) \bmod 3 = 0$ and $(c-b) \bmod 3 = 0$. Hence: $(c-a) \bmod 3 = (c-b+b-a) \bmod 3 = ((c-b) \bmod 3) + ((b-a) \bmod 3) = 0$. It follows that $a \preceq c$.

Anti-symmetry: Assume $a \preceq b$ and $b \preceq a$ for some $a, b \in \mathbb{N}$. Because $a \leq b$ and $b \leq a$, it follows that $a = b$.

(ii) This is not true. For example, there exists no greatest lower bound $0 \sqcap 1$: $p = 0$ is the only element for which $p \preceq 0$, but $0 \preceq 1$ does not hold.

(iii) This is true. Assume $a \preceq b$ for some $a, b \in \mathbb{N}$. Because $a \leq b$, it is $a+2 \leq b+2$. Further, it is $(b + 2 - (a + 2)) \bmod 3 = (b - a) \bmod 3 = 0$. Hence, it is $f(a) \preceq f(b)$.

(iv) This is true. Assume $a \preceq b$ for some $a, b \in \mathbb{N}$. Because $a \leq b$, it is $2a \leq 2b$. Further, it is $(2b-2a) \bmod 3 = ((b-a)+(b-a)) \bmod 3 = ((b-a) \bmod 3) + ((b-a) \bmod 3) = 0$. Hence, it is $f(a) \preceq f(b)$.

Solution 2.

a) Not a complete lattice. For example, $b \sqcup c$ does not exist.

b) Not a complete lattice. For example, $a \sqcap b$ does not exist.

c) This is a complete lattice.

d) Not a complete lattice. For example, $a \sqcap d$ does not exist.

Solution 3.

a) $\sqsubseteq = \{(a,a),(b,b),(c,c),(d,d),(a,b),(a,c),(b,d),(c,d),(a,d)\}$

Note: One needs to consider the reflexive transitive closure of the Hasse diagram. In particular, don't forget to include (a,d).

b) *Function f*: This function is monotone. The fixpoints and post-fixpoints are $\mathrm{Fix}(f) = \mathrm{Red}(f) = \{a,d\}$. The least fixpoint is a.

Function g: This function is not monotone. For example, it is $b \sqsubseteq d$, but $g(b) = b \not\sqsubseteq c = g(d)$. The fixpoints are $\mathrm{Fix}(g) = \{b,c\}$. The post-fixpoints are $\mathrm{Red}(g) = \{b,c,d\}$. There does not exist a least fixpoint.

Function h: This function is not monotone. For example, it is $c \sqsubseteq d$, but $h(c) = c \not\sqsubseteq b = h(d)$. The fixpoints are $\mathrm{Fix}(h) = \{c\}$. The post-fixpoints are $\mathrm{Red}(h) = \{b,c,d\}$. The least fixpoint is c.

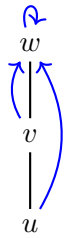
Solution 4.

a) First, f' neither approximates g' or h', because $f'(a) = c \not\sqsupseteq d = g'(a)$ and $f'(c) = c \not\sqsupseteq d = h'(c)$.

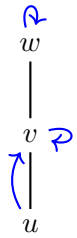
Next, note that g' always outputs the greatest element d. Hence, it approximates *any* function $A \to A$, in particular f' and h'.

Finally, h' neither approximates f' or g', because $h'(b) = c \not\sqsupseteq d = f'(b) = g'(b)$.

b) See the functions below. For (i), the depicted k is the only possible solution. For (ii), k' does not approximate any of f', g', h' due to $k'(v) = v$.



(i) Function k



(ii) Function k'

Exercise 01 - Solution

Documentation and Contracts

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) The current implementation implicitly assumes that the string returned by `getId` does not contain the hyphen character. Consider the following implementation:

```
public class MyNode implements Node {
    private String id;

    public MyNode(String id) {
        this.id = id;
    }

    public String getId() {
        return this.id;
    }
}
```

Now, consider the following code snippet:

```
MyNode a = MyNode("a");
MyNode bc = MyNode("b-c");
MyNode ab = MyNode("a-b");
MyNode c = MyNode("c");

// create more nodes and edges
// create ShortestPathsManager

List<Node> p1 = manager.getShortestPath(a, bc);
List<Node> p2 = manager.getShortestPath(ab, c);
```

In both calls to `getShortestPath`, the computed key is "a-b-c". Hence, the second call will incorrectly retrieve and return the cached result from the first call.

b) There are various options to fix the issue. One solution is to use a nested map `Map<String, Map<String, List<Node>>>` for the cache and perform a two-stage lookup of the form `cache.get(source.getId()).get(target.getId())`.

As a second solution, we can exchange the key type by a `Tuple` type separating the two identifiers.

If we don't want to change the type of `cache`, we could alternatively escape the hyphen character after calling `getId` (replace "\" by "\\", and "-" by "\-").

c) List of properties:

- Are the arguments allowed to be null?
- What is the return value if there is no shortest path (if `source` and `target` are disconnected in the graph)?
- What is the return value if `source` and `target` are equal?
- Does the returned list include `source` and `target`?
- If there are multiple shortest paths, which one is returned?
- What happens if `source` or `target` are not contained in the current graph?
- The caching behavior of the method.

d) We show an example documentation below. Note that there are different solutions and there is no unique "correct" behavior of the method. What's important is that the documentation covers the essential properties. Whether a property is essential is determined by the programmer: For example, the caching behavior may be considered incidental and subject to future change.

```
/**
 * Returns a path between nodes {@code source} and {@code target} with
 * minimal total distance in the graph. If there are multiple shortest
 * paths, any such path is returned. The shortest path for a specific
 * source and target is only computed at the first call to
 * {@code getShortestPath} and cached for subsequent calls.
 *
 * @param source  A non-null node contained in the graph
 * @param target  A non-null node contained in the graph
 * @return The shortest path encoded as the list of nodes along the path,
 *         excluding {@code source} and {@code target}. The return value
 *         is an empty list if {@code source == target}. The return value
 *         is {@code null} if {@code source} and {@code target} are
 *         disconnected in the graph.
 * @throws NodeNotFoundException if {@code source} or {@code target} are not
 *         contained in the current graph.
 */
public List<Node> getShortestPath(Node source, Node target) { ... }
```

Solution 2.

```
public class Player {
    private int x, y;
    public final int n;

    // invariant: 0 <= x && x < n && 0 <= y && y < n
```

```
    // requires: 0 <= x && x < n
    // requires: 0 <= y && y < n
    public Player(int x, int y, int n) {
        this.x = x;
        this.y = y;
        this.n = n;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }


    // ensures: old(getX()) == 0 => getX() == n - 1
    // ensures: old(getX()) > 0 => getX() == old(getX()) - 1
    // ensures: getY() == old(getY())
    public void moveLeft() {
        x = x - 1;
        if (x < 0) {
            x = n - 1;
        }
    }

    ...
}
```

Note 1: The constraint `n > 0` required for non-emptiness of the board is implied by the invariant and precondition. Hence, we do not need to state it explicitly.

Note 2: The postcondition refers to the methods `getX` and `getY`. This is only possible because they are side-effect free. Alternatively, we could refer to the fields `x` and `y` directly. However, because these are private, they are not directly accessible by clients. Generally, contracts for public methods should not refer to private fields.

Solution 3.

```
public class Bag {
    private int[] elems;
    private int count;

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(elems != null);
        Contract.Invariant(0 < elems.Length);
        Contract.Invariant(0 <= count && count <= elems.Length);
    }

    public Bag(int[] initialElements) {
        Contract.Requires(initialElements != null);
        Contract.Requires(0 < initialElements.Length);
        ...
    }

    public Bag(int[] initialElements, int start, int howMany) {
        Contract.Requires(0 <= start);
        Contract.Requires(0 < howMany);
        Contract.Requires(initialElements != null);
        Contract.Requires(start + howMany <= initialElements.Length);
        ...
    }

    [Pure]
    public int Count() { ... }

    [Pure]
    public int[] GetElements() { ... }

    public int RemoveMin() {
        Contract.Requires(0 < Count());
        ...
    }

    public void Add(int x) {
        Contract.Ensures(Count() == Contract.OldValue(Count()) + 1);
        Contract.Ensures(GetElements()[Contract.OldValue(Count())] == x)
        Contract.Ensures(!Contract.OldValue(Count() == GetElements().Length) ||
            GetElements().Length == 2*Contract.OldValue(GetElements().Length))
        Contract.Ensures(Contract.ForAll(0, Contract.OldValue(Count()),
            i => GetElements()[i] == Contract.OldValue(GetElements()[i])))
        ...
    }
}
```

Note: `elems.Length` must be strictly positive for `Add` to work correctly. We hence add according constraints to the invariant and the constructor preconditions.

Exercise 11 - Solution

Static Alloy Models

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) Figure 1 shows a UML-inspired class diagram modeling the university system. Undergraduate and graduate students are modeled as subclasses of Student. The Student class has different fields to maintain pointers to classmates, the major subject, etc. In this informal model, we do not represent many of the constraints stated by the system description.

b) See the Alloy model below.

```
abstract sig Bool{}
one sig True extends Bool{}
one sig False extends Bool{}

abstract sig Student{
  id: one ID,
  major: one Major,
  university: lone University,
  isLegal: one Bool,
  classmates: set Student,
}

sig Graduate extends Student{}
sig Undergraduate extends Student{}
sig ID{}
sig Major{}
sig University{}

fact unique_ids {
  all disj s, t: Student | s.id != t.id
}

fact no_id_without_student {
  all i: ID | one s: Student | s.id = i
}

```

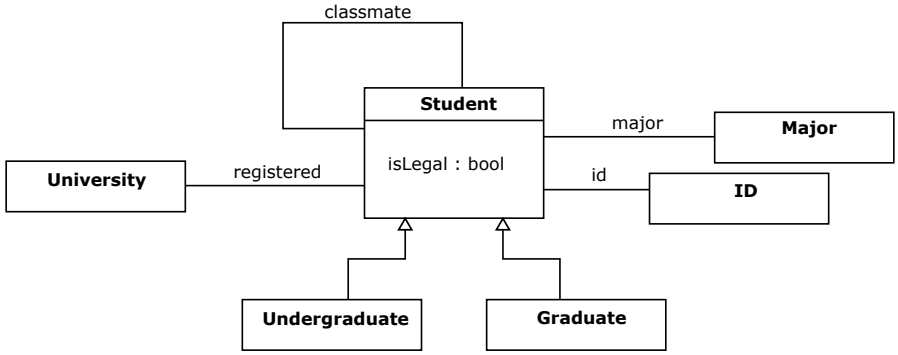


Figure 1: Class diagram of the university system.

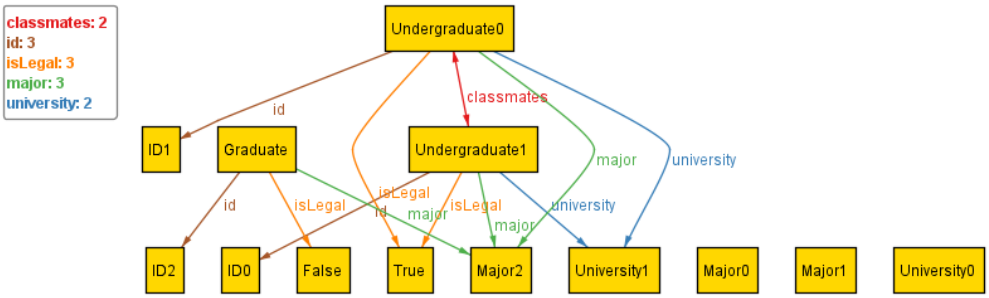


Figure 2: Example instance produced by Alloy.

```
27  fact legal_in_university {
28    all s: Student | (s.university != none) iff (s.isLegal = True)
29  }
30
31  fact classmates_have_same_major_and_uni {
32    all disj s, t: Student | (s.major = t.major and s.university = t.university and
33    (s in Undergraduate and t in Undergraduate
34    or s in Graduate and t in Graduate)) iff (s in t.classmates)
35  }
36
37  fact no_self_classmate {
38    all s: Student | s not in s.classmates
39  }
40
41  pred show {}
42
43  run show for exactly 2 University, exactly 3 Major, exactly 3 Student, exactly 3 ID
```

Notes:

- Making the `Student` signature **`abstract`** enforces every student to be either `Graduate` or `Undergraduate`.

- We do not specify any fields on the `ID`, `Major` and `University` signatures to abstract actual class implementations.
- Instead of defining booleans from scratch, you could use "**open** `util/boolean`" to load a boolean utility library. [1]
- Figure 2 shows an example instance produced by Alloy. You can use the "Next" button in Alloy to generate different instances.

Solution 2.

a) The model has a unique solution within the scope of 2 but 1 ETHBusStation as shown below. In this instance, all of ETHHonggerberg, HaldeneggLeft, Haldenegg-Right, Polyterrasse are the same bus station, whose `next` field points to itself.

b) See the modified Alloy model below for one possible solution.

```
1   abstract sig ETHBusStation {
2     next: set ETHBusStation
3   }
4
5   one sig Polyterrasse, HaldeneggRight, HaldeneggLeft,
6    ETHHonggerberg extends ETHBusStation {}
7
8   some sig ETHBus {
9     station: lone ETHBusStation
10  }
11
12  fact {
13    no (ETHHonggerberg.next - HaldeneggRight)
14    all s: ETHBusStation | ETHBusStation in s.^next and one s.next
15    all disj b1, b2: ETHBus | b1.station != b2.station
16  }
17
18  pred show {}
```

Note: Specification (iv) is satisfied in the original model. Line 15 in the original model includes the case `b1 = b2`, which precludes the existence of any ETHBus, hence violating (v).

[1] The library's Alloy model can be found in: `https://github.com/AlloyTools/org.alloytools.alloy/blob/master/org.alloytools.alloy.core/src/main/resources/models/util/boolean.als`

Solution 3.

a) See the Alloy model below. The Alloy analyzer cannot find a counterexample for the provided scope. Note that while this provides some indication that the claim is true, it is not a proof.

```
1   sig Person {
2     loves: set Person
3   }
4
5   one sig Me in Person {}
6
7   one sig Baby in Person {}
8
9   fact song {
10    (all p: Person | Baby in p.loves) and Baby.loves = Me
11  }
12
13  assert claim {
14    Me = Baby
15  }
16
17  check claim for 5
```

b) See the modification of Line 10 as shown below. Now, the Alloy analyzer finds a counterexample, demonstrating that the implication is not true.

```
10  (all p: Person | Baby in p.loves) and Baby.loves = Me + Baby
```

Exercise 07 - Solution

Widening and Pointer Analysis

Rigorous Software Engineering, ETH Zurich

Solution 1.

a) First, we introduce a phantom label $4'$ pointing at the state immediately after line 4 (to simplify joining the control flow in line 2).

The iterations are shown below. Note how in the fifth iteration ($\overset{*}{\rightsquigarrow}$), the interval for y at line 2 is widened.

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$\bot$	$\bot$
3	$\bot$	$\bot$
4	$\bot$	$\bot$
$4'$	$\bot$	$\bot$
5	$\bot$	$\bot$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, 0]$
3	$\bot$	$\bot$
4	$\bot$	$\bot$
$4'$	$\bot$	$\bot$
5	$\bot$	$\bot$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, 0]$
3	$[1, 100]$	$[0, 0]$
4	$\bot$	$\bot$
$4'$	$\bot$	$\bot$
5	$[-\infty, \infty]$	$[0, 0]$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, 0]$
3	$[1, 100]$	$[0, 0]$
4	$[1, 100]$	$[1, 100]$
$4'$	$\bot$	$\bot$
5	$[-\infty, \infty]$	$[0, 0]$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, 0]$
3	$[1, 100]$	$[0, 0]$
4	$[1, 100]$	$[1, 100]$
$4'$	$[0, 99]$	$[1, 100]$
5	$[-\infty, \infty]$	$[0, 0]$

$\overset{*}{\rightsquigarrow}$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, \infty]$
3	$[1, 100]$	$[0, 0]$
4	$[1, 100]$	$[1, 100]$
$4'$	$[0, 99]$	$[1, 100]$
5	$[-\infty, \infty]$	$[0, 0]$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, \infty]$
3	$[1, 100]$	$[0, \infty]$
4	$[1, 100]$	$[1, 100]$
$4'$	$[0, 99]$	$[1, 100]$
5	$[-\infty, \infty]$	$[0, \infty]$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, \infty]$
3	$[1, 100]$	$[0, \infty]$
4	$[1, 100]$	$[1, \infty]$
$4'$	$[0, 99]$	$[1, 100]$
5	$[-\infty, \infty]$	$[0, \infty]$

$\rightsquigarrow$

	x	y
1	$[-\infty, \infty]$	$[-\infty, \infty]$
2	$[-\infty, \infty]$	$[0, \infty]$
3	$[1, 100]$	$[0, \infty]$
4	$[1, 100]$	$[1, \infty]$
$4'$	$[0, 99]$	$[1, \infty]$
5	$[-\infty, \infty]$	$[0, \infty]$

b) Consider the following program. Analysis without widening computes the interval $[4, 4]$ for `x` at line 4. Analysis with widening converges faster, but computes the less precise interval $[4, \infty]$ for `x` at line 4.

```
1:  x := 1
2:  while (x <= 3)
3:      x := x + 1
4:
```

Solution 2. In the following, let (m_p, m_h) be the tuple of the pointer map m_p and the heap map m_h of the abstract state at the current label.

a) Assuming the allocation occurs at label l, we can directly assign l to `x` in the pointer map. Formally:

$$[\![\texttt{x := alloc}]\!](m_p, m_h) = (m_p[\texttt{x} \mapsto \{l\}],\ m_h).$$

b) We have to compute the intersection of the points-to sets of `x` and `y`:

$$[\![\texttt{x == y}]\!](m_p, m_h) = (m_p[\texttt{x} \mapsto m_p(\texttt{x}) \cap m_p(\texttt{y}),\ \texttt{y} \mapsto m_p(\texttt{x}) \cap m_p(\texttt{y})],\ m_h).$$

c) As you have seen in the lecture, we need to perform a *weak update* here. The reason is that the abstract objects A which `x` points to may be an over-approximation of the concrete objects receiving the update at runtime.

Intuitively, we have to union the points-to sets of `y` and `x.f` in the heap map. Formally, the transformer is

$$[\![\texttt{x.f := y}]\!](m_p, m_h) = (m_p, m'_h),$$

where m'_h is defined as follows:

$$m'_h(o.g) = \begin{cases} m_h(o.g) & \text{if } g \neq \texttt{f}, \\ m_h(o.\texttt{f}) \cup m_p(\texttt{y}) & \text{else if } o \in m_p(\texttt{x}), \\ m_h(o.\texttt{f}) & \text{otherwise} \end{cases}$$

d) In contrast to heap stores, we can use *strong updates* for local variable stores. The target receiving the update (the local variable) is known exactly at analysis time.

We first have to find all possible abstract objects o in the points-to set of `y` and union the points-to sets of their field `f`. Then, we assign the resulting set to `x`, discarding the old set (strong update). Formally:

$$[\![\texttt{x := y.f}]\!](m_p, m_h) = \Big(m_p\Big[x \mapsto \bigcup_{o \in m_p(\texttt{y})} m_h(o.\texttt{f})\Big],\ m_h\Big)$$

Note: Strong updates also apply to direct pointer assignments (`x := y`), with a simplified variant of the transformer above. The rule in (a) is also a strong update.

Solution 3.

a) First, we introduce a phantom label $8'$ pointing at the state immediately after line 8. There are two allocation sites a1 and a5 in lines 1 and 5, respectively.

The three global steps required for convergence of the while loop are shown below (we don't show the many intermediate iterations). Note that $\top$ includes null.

	t	c	n	heap
1	$\top$	$\top$	$\top$	
2	$\top$	a1	$\top$	
3	a1	a1	$\top$	
4	a1	a1	$\top$	
5	a1	a1	$\top$	
6	a1	a1	a5	
7	a1	a1	a5	a1.f $\mapsto$ {a5}
8	a1	a5	a5	a1.f $\mapsto$ {a5}
$8'$	a1	a5	a5	a1.f $\mapsto$ {a5}
9	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}
10	a1	a1, a5	$\top$	a1.f $\mapsto$ {a1, a5}, a5.f $\mapsto$ {a1}

$\rightsquigarrow$

	t	c	n	heap
1	$\top$	$\top$	$\top$	
2	$\top$	a1	$\top$	
3	a1	a1	$\top$	
4	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}
5	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}
6	a1	a1, a5	a5	a1.f $\mapsto$ {a5}
7	a1	a1, a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
8	a1	a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
$8'$	a1	a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
9	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
10	a1	a1, a5	$\top$	a1.f $\mapsto$ {a1, a5}, a5.f $\mapsto$ {a1, a5}

	t	c	n	heap
1	$\top$	$\top$	$\top$	
2	$\top$	a1	$\top$	
3	a1	a1	$\top$	
4	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
5	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
$\rightsquigarrow$ 6	a1	a1, a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
7	a1	a1, a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
8	a1	a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
8′	a1	a5	a5	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
9	a1	a1, a5	$\top$	a1.f $\mapsto$ {a5}, a5.f $\mapsto$ {a5}
10	a1	a1, a5	$\top$	a1.f $\mapsto$ {a1, a5}, a5.f $\mapsto$ {a1, a5}

b) The assertion in line 7 can be proven. According to the abstract points-to state at line 8, the points-to sets {a1} and {a5} for `t` resp. `c` are disjoint and the two pointers are hence guaranteed to not alias.

In contrast, the assertion in line 10 cannot be proven. The points-to sets of `t` and `c` at line 10 both include the abstract object a1 $\neq$ **`null`**.

Because this is a *may-point-to* analysis, we can not automatically conclude that `c` and `t` must alias in line 10. Indeed, this is not true here: If the argument `count` is 0, the pointers alias because the loop body is not executed. However, if `count` is 1, the pointers don't alias due to the assignment in line 7.