ASE VU Exercise 1

Duedate: November 8, 2023

Florian Freitag

Exercise 1: Abstract Interpretation - Parity Domain

Consider the parity domain which has abstract elements $\{\bot, Even, Odd, \top\}$ where $Even$ is the set of all even integers including 0 and Odd is the set of all odd integers.

Using the parity domain, design abstract transformers for the statements and expressions in the function f below and define a join operator. Then perform Abstract Interpretation on f. To this end construct the (abstract) control flow graph, where the nodes are the abstract states and the edges are the statements of f. Finally establish that the asserted condition holds.

```
0  void f(int n) {
1    int p = 13;
2    int m = 101;
3    if (n % 2 == 0)  {
4      if (m > 0) {
5        p = p + n;
6        m = m - 1;
7        goto 4;
8      }
9    }
10   p = p - 1;
11   assert p % 2 == 0;
12 }
```

Solution:

First, lets define our join operator $\triangledown$: The operator should be the u with corners

$$\begin{aligned}
\bot \triangledown \{\bot, Even, Odd, \top\} &= \bot \\
\top \triangledown \{Even, Odd, \top\} &= \top \\
Even \triangledown Odd &= \top \\
Even \triangledown Even &= Even \\
Odd \triangledown Odd &= Odd
\end{aligned}$$

So this could also be a table

This definition doesn't cover all cases, however since this operator is comutative all possible cases can be derived by switching the operands.

Next, let's define the $+$ and $-$ operation for the abstract transformer. Since both operations behave the same we will define them once and note them wiht the $\pm$ operator. Even though it may seem strage at first but the $\pm$ operations are comutative in this domain.

$$\begin{aligned}
\bot \pm \{\bot, Even, Odd, \top\} &= \bot \\
\top \pm \{Even, Odd, \top\} &= \top \\
Even \pm Even &= Even \\
Odd \pm Odd &= Even \\
Even \pm Odd &= Odd
\end{aligned}$$

Finally we can use the transformer to interpret the program and verify the condition in line 11:

$$pc = 0 \quad n = \bot \quad m = \bot \quad p = \bot$$
$$\downarrow$$
`0: int n`
$$\downarrow$$
$$pc = 1 \quad n = \top \quad m = \bot \quad p = \bot$$
$$\downarrow$$
`1: int p = 13`
$$\downarrow$$
$$pc = 2 \quad n = \top \quad m = \bot \quad p = Odd$$
$$\downarrow$$
`2: int m = 101`
$$\downarrow$$
$$pc = 3 \quad n = \top \quad m = Odd \quad p = Odd$$
$$\downarrow$$
`3: if (n % 2 == 0)`
$$\downarrow$$
$$(pc = 4 \quad n = Even \quad m = Odd \quad p = Odd) \quad (pc = 10 \quad n = Odd \quad m = Odd \quad p = Odd)$$
$$\searrow$$
`4: if (m > 0)`
$$\downarrow$$
$$(pc = 5 \quad n = Even \quad m = Odd \quad p = Odd) \quad (pc = 10 \quad n = Even \quad m = Odd \quad p = Odd)$$
$$\searrow$$
`5: p = p + n`
$$\downarrow$$
$$pc = 6 \quad n = Even \quad m = Odd \quad p = Odd$$
$$\downarrow$$
`6: m = m - 1`
$$\downarrow$$
$$pc = 7 \quad n = Even \quad m = Even \quad p = Odd$$
$$\downarrow$$
`7: goto 4`
$$\searrow$$
$$(pc = 4 \quad n = Even \quad m = Odd \quad p = Odd) \quad (pc = 4 \quad n = Even \quad m = Even \quad p = Odd)$$
$$\searrow \quad \swarrow$$
Join
$$\downarrow$$
$$pc = 4 \quad n = Even \quad m = \top \quad p = Odd$$
$$\downarrow$$

continued on next page...

$$\downarrow$$

`4: if (m > 0)`

$$\downarrow$$

$$(pc = 5 \quad n = Even \quad m = \top \quad p = Odd) \quad (pc = 10 \quad n = Even \quad m = \top \quad p = Odd)$$

$$\searrow$$

`5: p = p + n`

$$\downarrow$$

$$pc = 6 \quad n = Even \quad m = \top \quad p = Odd$$

$$\downarrow$$

`6: m = m - 1`

$$\downarrow$$

$$pc = 7 \quad n = Even \quad m = \top \quad p = Odd$$

$$\downarrow$$

`7: goto 4`

$$\downarrow$$

$$pc = 4 \quad n = Even \quad m = \top \quad p = Odd$$

$$\begin{pmatrix} pc = 10 \\ n = Odd \\ m = Odd \\ p = Odd \end{pmatrix} \quad \begin{pmatrix} pc = 10 \\ n = Even \\ m = Odd \\ p = Odd \end{pmatrix} \quad \begin{pmatrix} pc = 10 \\ n = Even \\ m = \top \\ p = Odd \end{pmatrix}$$

$$\searrow \quad \downarrow \quad \swarrow$$

Join

$$\downarrow$$

$$pc = 10 \quad n = \top \quad m = \top \quad p = Odd$$

$$\downarrow$$

`10: p - 1`

$$\downarrow$$

$$pc = 11 \quad n = \top \quad m = \top \quad p = Even$$

With that we can see that the assertion on line 11 always holds. The final states of the interpretation are:

pc	0	1	2	3	4	5	6	7	$8-10$	11
n	$\bot$	$\top$	$\top$	$\top$	$Even$	$Even$	$Even$	$Even$	$\top$	$\top$
m	$\bot$	$\bot$	$\bot$	Odd	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
p	$\bot$	$\bot$	Odd	Odd	Odd	Odd	Odd	Odd	Odd	$Even$

Exercise 2: Abstract Interpretation - Comparison of Domains

Give either a program that can be verified with an abstract interpreter using the parity domain but not with the interval domain or show that such a program cannot exist. Repeat this exercise, but use the sign domain and the interval domain respectively.

Solution:

```
0  void p1(int n) {
1    int a = 0;
2    if (n > 0)  {
3      n = n - 1;
4      a = a + 2;
5      goto 2;
6    }
7    assert a % 2 == 0;
8  }
```

Program `p1` can be verfied with the parity domain and it is easy to see that `a`, once initialized to *Even* would remain that state over rest of the interpretation. In the interval domain `a` would be widend to $[0, +\infty]$ and would therefore use the relevant information that `a` remains even.

For the sign domain and interval domain, no such program can be found as the interval domain can express all states of the sign domain and many more. For example we could come up with a mapping from the sign domain to the interval domain:

$$\begin{aligned} \bot &\Rightarrow \bot \\ \top &\Rightarrow [-\infty, +\infty] \\ - &\Rightarrow [< 0, < 0] \\ + &\Rightarrow [> 0, > 0] \\ 0 &\Rightarrow [0, 0] \end{aligned}$$

Maybe state infinity here than just smaller zero

In contrast no such mapping would be possible from the parity domain to the interval domain, even though the interval domain is infinite and the parity is not.

Exercise 3: Abstract Interpretation - Interval Domain

Consider again the function Foo from the lecture:

```
0  void Foo(int i) {
1    int x = 5;
2    int y = 7;
3    if (i >= 0) {
4      y = y + 1;
5      i = i - 1;
6      goto 3;
7    }
8    assert 0 <= y - x;
9  }
```

Give a modified interval domain INT' s.t. an Abstract Interpreter using INT' verifies Foo without using widening. Similarily to the previous example, give the (abstract) control flow graph of the Abstract Interpretation run.

After that give a new program P with an asserted property that cannot be verified with an Abstract Interpreter that uses INT' but can be verified with an Abstract Interpreter using the standard interval domain from the lecture. Again you are not allowed to use widening.

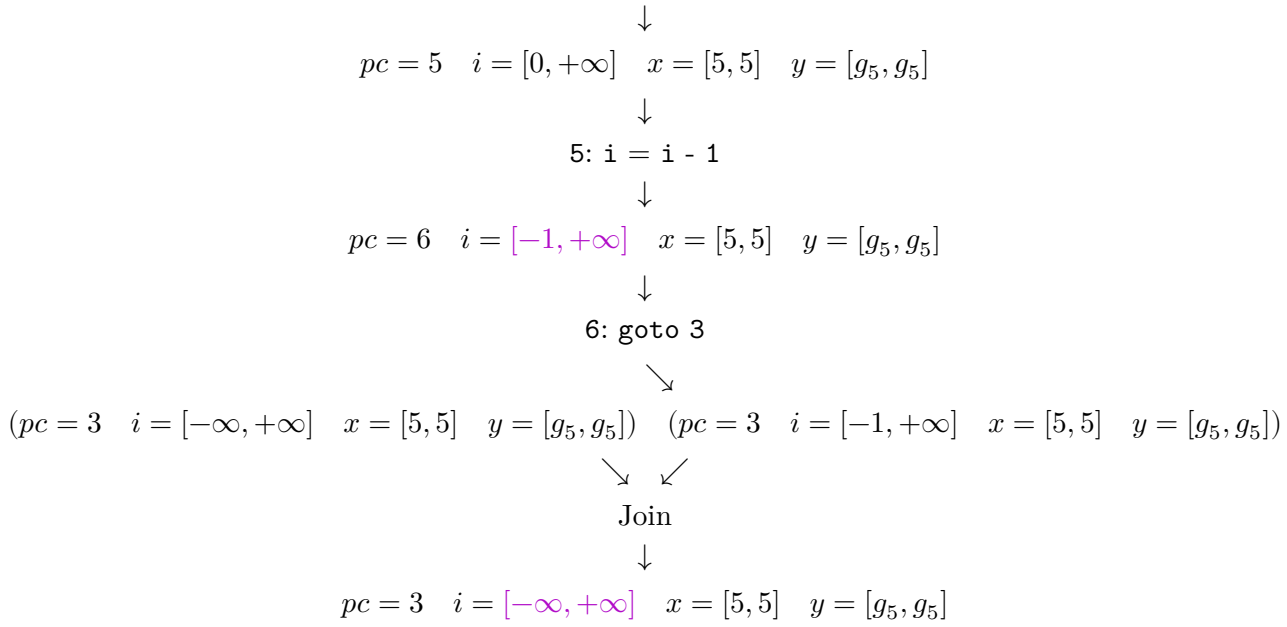
Solution:

Extending the Interval domain by a new atom g_5 indicating that a bound is greater than five leads to the following control flow:

So the others have listed all the possible values listed instead of extending it. But the basic idea is the same

$$pc = 0 \quad i = \bot \quad x = \bot \quad y = \bot$$
$$\downarrow$$
$$\texttt{0: int i}$$
$$\downarrow$$
$$pc = 1 \quad i = [-\infty, +\infty] \quad x = \bot \quad y = \bot$$
$$\downarrow$$
$$\texttt{1: int x = 5}$$
$$\downarrow$$
$$pc = 2 \quad i = [-\infty, +\infty] \quad x = [5, 5] \quad y = \bot$$
$$\downarrow$$
$$\texttt{2: int y = 7}$$
$$\downarrow$$
$$pc = 3 \quad i = [-\infty, +\infty] \quad x = [5, 5] \quad y = [g_5, g_5]$$
$$\downarrow$$
$$\texttt{3: if (i >= 0)}$$
$$\downarrow$$
$$(pc = 4 \quad i = [0, +\infty] \quad x = [5, 5] \quad y = [g_5, g_5]) \quad (pc = 8 \quad i = [-\infty, -1] \quad x = [5, 5] \quad y = [g_5, g_5])$$
$$\searrow$$
$$\texttt{4: y = y + 1}$$
$$\downarrow$$

continued on the next page...



The only time we reach line 8 `y` is greater than 5 and `x` is exaclty 5 therefore the assertion will always hold true.

Now, let's consider the following program `P`:

```
0  void P(int n) {
1    a = 2;
2    if (n >= 0) {
3      n = n - 1;
4      a = a - 1;
5    }
6    assert a < 3;
7  }
```

So my solution here doesn't work because the normal Interval domain because we were not allowed to use widening even with the Interval domain.

However, my domain is just not good enough for this subexercise.

With our modified domain INT' it is not possible to verify the program as the interpretation would not terminate. This is because in each iteration of the loop a would get a new interval (with both barries lower by one as than before) and the interpretation would never reach a fixed point. The newly inserted atom g_5 doesn't help in that case.

With the domain from the lecture where widening is allowed the program can be verified as a would ultimatly get the interval $[-\infty, 2]$ which is always less than 3.

Exercise 4: Abstract Interpretation - Soundness of Abstractions

Consider imperative programs defined over integer variables with the following instructions: **declare** v, **assign** $v\,e$, **mod3** v and **assert** v. We call v, v_1, v_2 integer variables and e an expression, where e follows the syntax $(v + c_1)$ or $(v_1 + v_2)$ or $(c_1 + c_2)$ and $c_1, c_2 \in \mathbb{Z}$.
The semantics of this programming language follows closely the semantics of the programming language C (assuming C uses mathematical integers, so no overflows occur). We have the following correspondence:

declare $v \Rightarrow$ int v
assign $v\,e \Rightarrow v = e$
mod3 $v \Rightarrow v = v\,\%\,3$
assert $v \Rightarrow$ assert $(v\,!= 0)$

Now that we defined the semantics of our programming language, let's define the abstract semantics. In the following we use the abstract domain $\mathcal{D} = \{\bot, \dot{0}, @, \bigcirc, \top\}$. Our abstract state is now either a mapping α from all program variables to their abstract value in $\mathcal{D}$ or the special error state Err :

$$\alpha(v) = \begin{cases} \bot & \text{if v is not declared} \\ \dot{0} & \text{if } v = 0 \\ @ & \text{if } v\%3 = 0 \text{ and } 0 < v < 1000 \\ \bigcirc & \text{if } v\%3 \neq 0 \\ \top & \text{otherwise} \end{cases}$$

We extend α s.t. it also computes the abstract value for constants and the abstract value for a (declared) expression. In the table below you can see the abstract value associated to expressions depending on the abstract values of their two operands.

$+$	$\dot{0}$	$@$	$\bigcirc$	$\top$
$\dot{0}$	$\dot{0}$	$@$	$\top$	$\top$
$@$	$@$	$\top$	$\bigcirc$	$\top$
$\bigcirc$	$\top$	$\bigcirc$	$\bigcirc$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

This is the unsound part

The abstract transformers defined in the following describe how statements of our programming language affect the abstract states. Let's define $f[a \mapsto b]$ as a function f' with $f'(a) = b$ and $f'(x) = f(x)$ for all $x \neq a$.

$$\alpha \xrightarrow{\text{assign } ve} \alpha[v \mapsto \alpha(e)]$$

$$\alpha \xrightarrow{\text{declare } v} \alpha[v \mapsto \top]$$

$$\alpha \xrightarrow{\text{mod } 3v} \begin{cases} \alpha & \text{if } \alpha(v) = \bigcirc \\ \alpha[v \mapsto \dot{0}] & \text{if } \alpha(v) = @ \\ \alpha[v \mapsto \top] & \text{otherwise} \end{cases}$$

$$\alpha \xrightarrow{\text{assert } v} \begin{cases} \alpha & \text{if } \alpha(v) \in \{@, \bigcirc\} \\ Err & \text{otherwise} \end{cases}$$

Show that the above defined abstraction is unsound by giving a program P in our programming language which would yield an error (a failing assertion) with the concrete semantics but for which the Abstract Interpretation terminates in a state different from the state Err. Provide all intermediate abstract states that are constructed when performing Abstract Interpretation on P.

Solution:

Consider the follwing program `P`:

```
0  declare a           // int a;
1  assign a (1 + 1)    // a = 1 + 1;
2  assign a (a + 1)    // a = a + 1;
3  mod3 a              // a = a % 3;
4  assert a            // assert (a != 0);
```

Both assigns could be merged to:
assign a (1 + 2)

It is obvious that the program would fail the assertion as the first three lines set `a` to 3, which then get's set to 0 be the modular operator and violates the assertion.

However, the abstract interpretation does not exit with Err. As a side note 1 evaluates to the abstract state $\bigcirc$ and $\bigcirc + \bigcirc = \bigcirc$.

$$\alpha = \{a : \bot\}$$
$$\downarrow$$
0: `declare a`
$$\downarrow$$
$$\alpha = \{a : \top\}$$
$$\downarrow$$
1: `assign a 1+1`
$$\downarrow$$
$$\alpha = \{a : \bigcirc\}$$
$$\downarrow$$
2: `assign a a+1`
$$\downarrow$$
$$\alpha = \{a : \bigcirc\}$$
$$\downarrow$$
3: `mod3 a`
$$\downarrow$$
$$\alpha = \{a : \bigcirc\}$$
$$\downarrow$$
4: `assert a`
$$\downarrow$$
$$\alpha = \{a : \bigcirc\}$$

Exercise 5: Abstract Interpretation - Bounded Model Checking

Consider the C function below and try to verify it by performing Bounded Model Checking. Use a loop unrolling bound of 2 and perform every step of Bounded Model Checking up until the Bit-Blasting step. Would a SAT solver invoked by the Bounded Model Checker finally output SAT or $UNSAT$? In case it would output SAT, give a satisfying assignment of the integer variables in the constraints constructed. In case it would output UNSAT, explain why such an assignment cannot exist. What can we learn from the output of the SAT solver about the safety of the program?

```
0  void sum(unsigned int n){
1    unsigned int i = 0;
2    unsigned int s = 0;
3    if (n < 10000) {
4      while(++i < 5) {
5        s += n;
6      }
7    }
8    assert(s <= 5 * n);
9  }
```

Solution:

After simplifying controllflow and loop unrolling we get the following adaptation:

```
0  void sum(n){
1    i = 0;
2    s = 0;
3    if (n < 10000) {
4      i = i + 1;
5      if (i < 5) {
6        s = s + n;
7        i = i + 1;
8        if (i < 5) {
9          s = s + n;
10         i = i + 1;
11         assume (i >= 5)
12       }
13     }
14   }
15
16   assert(s <= 5 * n);
17 }
```

continued on the next page...

This is now easily convertable to the following SSA:

```
0  void sum(n0){
1    i0 = 0;
2    s0 = 0;
3    if (n0 < 10000) {
4      i1 = i0 + 1;
5      if (i1 < 5) {
6        s1 = s0 + n0;
7        i2 = i1 + 1;
8        if (i2 < 5) {
9          s2 = s1 + n0;
10         i3 = i2 + 1;
11         assume (i3 >= 5)
12       }
13     }
14   }
15        <- The s2 here is wrong we need to merge all s0, s1 and s2 into one
16   assert(s2 <= 5 * n0);
17 }
```

And finally we can convert it to the following constraints:

$$
\begin{aligned}
i_0 &= 0 \wedge \\
s_0 &= 0 \wedge \\
i_1 &= i_0 + 1 \wedge \\
s_1 &= s_0 + n_0 \wedge \\
i_2 &= i_1 + 1 \wedge \\
s_2 &= s_1 + n_0 \wedge \\
i_3 &= i_2 + 1 \wedge \\
s_2 &> 5 * n_0
\end{aligned}
$$

The SAT solver would output *UNSAT* which we can see by expanding s_2 in the last subconstraint:

$$
\begin{aligned}
s_2 &> 5 * n_0 \\
s_1 + n_0 &> 5 * n_0 \\
s_0 + n_0 + n_0 &> 5 * n_0 \\
0 + n_0 + n_0 &> 5 * n_0
\end{aligned}
$$

Obviously $2 * n_0 > 5 * n_0$ is never true.