We asked you to look into two exercises at the end of the abstract interpretation lecture.

This document describes the corresponding abstract interpretation designs and answers the posed questions (hopefully in sufficient detail). Please read through these, and **please ask if there's anything – major or minor, concept or detail** – that you are unsure of.

Exercise 1

This exercise asked you to answer two different questions about the function f, defined below:

```
f(a, b, c):
      if (*):
            x = a + b
      else:
            x = a + c
      return x
```

The questions are:

- 1. Which variables' values are **definitely** used in f?
- 2. Which variables' values **might** be used in f?

(Note: the * condition means "random choice")

Consider a lattice with the following elements (element name on the left, definition on the right):

T — ***T**op*: *may or may not have been used (unknown)*
|
U — ***U**sed*: *definitely used*

What should the **abstract store (i.e., abstract state)** look like at the entry point of f? Nothing has yet been used. So, the abstract store is:

[x -> T, a -> T, b -> T, c -> T] Read this as: x has abstract type T, a has abstract type T, …

Note that the abstract store is a *mapping from each variable* to an element of the abstract domain. We'll have an abstract store for each *program point*, which we can define as the union of all the entry and exit points from the nodes in the control flow graph (CFG). This is similar to the concrete store (aka memory).

The next program point to consider is the if statement condition. Since this doesn't involve any of our variables (it is just *, the random choice operator), it inherits the stores from the first program point.

Now, we need to consider the two branches of the if statement. In the then branch, we have the statement:

```
x = a + b
```

Before this statement, we inherit the abstract state of the function entry point:

[x -> T, a -> T, b -> T, c -> T]

Now, we need a transfer statement for an assignment. For this analysis, the transfer statement for assignment has two parts: one for the left-hand-side (lhs) of the assignment, and another for the right-hand-side. The lhs transfer takes only one argument: the abstract state of the variable being defined. Here's the definition:

transfer_lhs(lhs):

lhs	**transfer_lhs(lhs)**
T	T
U	T

That is, after we assign a new value to a variable, it is always transferred to T (*possibly used*) because the new value hasn't been used. However, we don't have a lattice element that exactly represents that, so we choose the point that represents "unknown". A more precise analysis (or one intended to answer different questions) might include a "definitely not used" lattice element, but we don't need one to answer our questions.

The transfer function for the rhs is more complicated: it takes an arbitrary number of variables' abstract states: one for each variable in the expression on the rhs. Without loss of generality, however, we can consider each variable individually, because (unlike e.g. addition in the parity domain) the transfer of one variable is unaffected by the abstract values of the others. Here's the definition:

transfer_rhs(rhs):

rhs	**transfer_rhs(rhs)**
T	U
U	U

What does this say? A variable that we don't know has been used becomes used (because it was used!). A variable we already know is used stays used.

So, which functions do we apply? Because the lhs is x, we apply transfer_lhs(x), to transition the state to (note that this is a no-op):

[x -> T, a -> T, b -> T, c -> T]

Then, we apply transfer_rhs to each variable on the rhs (in a real implementation, we'd do that by recursively descending into the tree until we reach individual variable nodes):

transfer_rhs(a); transfer_rhs(b)

That produces this state, which is the state after the then branch:

[x -> T, a -> U, b -> U, c -> T]

Similarly, the state at the beginning of the else branch is the initial state:

[x -> T, a -> T, b -> T, c -> T]

Then, we apply:

transfer_lhs(x); transfer_rhs(a); transfer_rhs(c);

To produce:

[x -> T, a -> U, b -> T, c -> U]

That is the state after the else branch.

Then, we need to merge the then-branch state and the else-branch state. The two states are:

then: **[x -> T, a -> U, b -> U, c -> T]**
else: **[x -> T, a -> U, b -> T, c -> U]**

The question we are interested in answering determines how to merge. If we are interested in the question "which variables are **definitely** used?", then we want to take the least upper bound (lub) of each element in the state. That's because we're interested in variables that are used on *every* feasible path, so only those that are U on at least one path and T on none. For completeness, here's the definition of the lub function, which has two arguments (represented here as the first row and first column):

lub	T	U
T	T	T
U	T	U

Doing this for the two states above gives us this final state:

then: **[x -> T, a -> U, b -> U, c -> T]**
else: **[x -> T, a -> U, b -> T, c -> U]**
lub: **[x -> T, a -> U, b -> T, c -> T]**

Notice that b and c both go to T, because on at least one path they were unused. Therefore, the answer is “only a is definitely used”, because only a maps to U at the end of the function.

By contrast, if we want to answer “which variables **might** be used?”, then we want to use the greatest lower bound (glb), whose definition is:

glb	T	U
T	T	U
U	U	U

(Note the symmetry between these two definition tables)

Applying glb to the then and else states:

then: **[x -> T, a -> U, b -> U, c -> T]**
else: **[x -> T, a -> U, b -> T, c -> U]**
glb: **[x -> T, a -> U, b -> U, c -> U]**

That lets us conclude that “a, b, and c might be used” by the end of the function.

Exercise 2

This exercise asked you to think about why an abstract domain containing the following abstract values terminates on the function g (defined below):

odd	*x % 2 == 1*
even2	*x % 2 == 0*
even4	*x % 4 == 0*
is2	*x == 2*

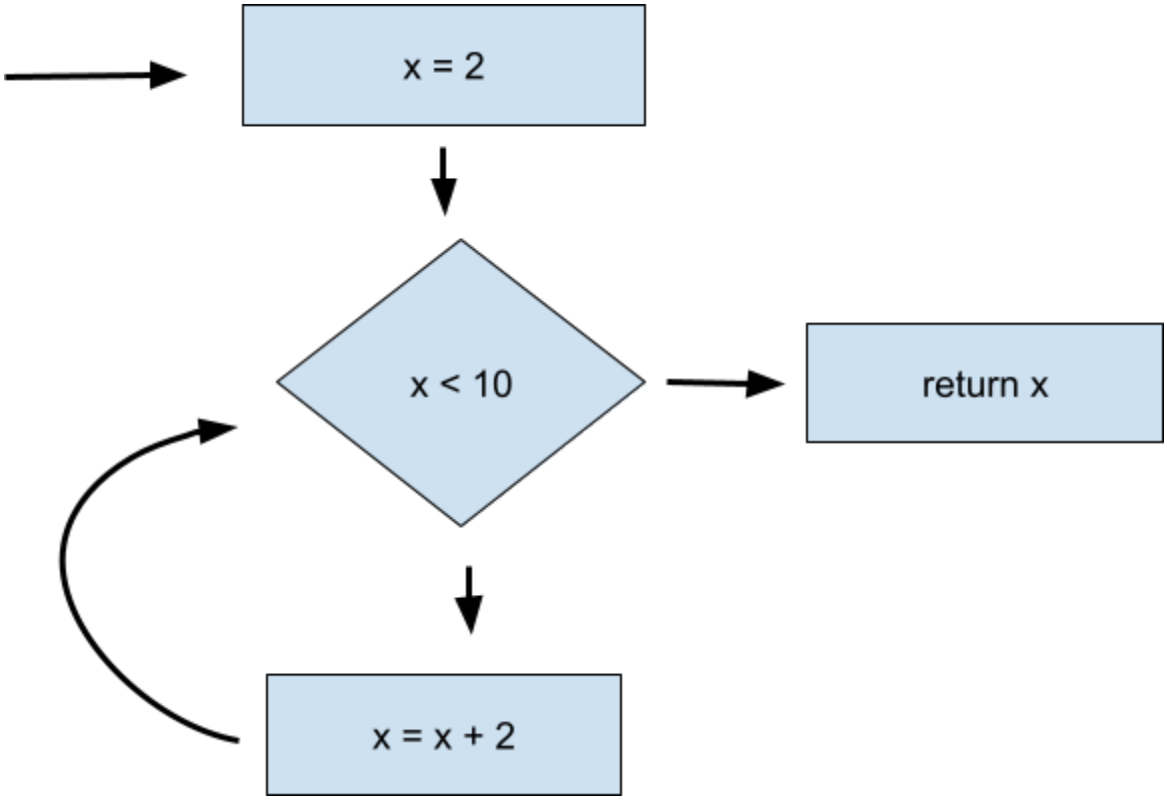
Implicit in the question is that we also need to have a top and bottom lattice element.

Here's the definition of g, the function we want to look at:

```
g():
      x = 2
      while (x < 10):
            x = x + 2
      return x
```

Note that g only has one variable. That means our abstract stores will only have one entry at each program location.

How many abstract stores are there (i.e., what program locations are there)? To figure that out, we need a CFG. The CFG for g looks like this:



On entry to the function, we have this store:

[x -> ⊥]

That is, x is undefined. (If x was a formal parameter of function g, the initial store would be [x -> T]; convince yourself that this makes sense and understand why.)

Following the CFG edges, we can now process the node “x = 2”. Because our abstract domain has an element for “is exactly 2”, we can conclude that the store after processing that node is:

[x -> is2]

At this point, we have to enter the test node. It has two incoming CFG edges, so we have to merge the stores for each. We don’t know anything about the other edge (the one that comes from the end of the while-loop), so we use a store that maps x to bottom, and then use the lub:

[x -> ⊥] lub [x -> is2] = [x -> is2]

So for now, we’ll enter the loop with the store **[x -> is2]**.

At this point, we can process the statement `x = x + 2`. “2” is abstracted as “is2”. We can then consult the transfer function for +:

+	T	odd	even2	even4	is2	⊥
T	T	T	T	T	T	⊥
odd	T	even2	odd	odd	odd	⊥
even2	T	odd	even2	even2	even2	⊥
even4	T	odd	even2	even4	even2	⊥
is2	T	odd	even2	even2	even4	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

(Look at the table and convince yourself it's correct)

We're in the case where both arguments to the + are abstracted as is2, so we now have the store:

[x -> even4]

Now, we have to follow the CFG edge back to the start of the loop, and merge it with our store from before:

[x -> even4] lub [x -> is2] = [x -> even2]

Following a similar process, we reprocess "x = x + 2" in this new context. The transfer function for +(even2, is2) induces this store afterward:

[x -> even2]

Since that's different from our last time around the loop, we have to process the loop again. The transfer function for +(even2, is2) still results in the store:

[x -> even2]

So, we can stop processing the loop (we've reached a fixpoint). Then, we process the last CFG edge and conclude that at the end, the result store is:

[x -> even2]

Additional exercises

Test your understanding by working through the following three exercises. (I added additional questions, so you can test whether you could generalize your understanding to a different program.)

Given an abstract interpretation design, here are the steps to apply it:

1. Determine the set of program points for which an abstract store is needed.
2. Create an **initial abstract store** for each program point, **before** doing **any interpretation** of the code!
    a. At this point, you only know:
        i. The number of program points
        ii. The number and kind of variables that need to be tracked
    b. At this point, you do not know anything about the statements in between the program points.
3. Abstractly interpret the code (statement by statement) and update the abstract stores.

Abstract domain

For all three exercises, we consider the following abstract domain:
* **T**: **T**op (unknown)
* **O**: **O**dd (x % 2 == 1)
* **E**: **E**ven (x % 2 == 0)
*⊥: Bottom (undefined)

Questions

- What is an adequate lattice for this domain?
  (Recall that (1) the lattice must include all types and (2) the edges define subtype relationships. Note that the edges do not correspond to the transfer functions, defined over the abstract domain.
- Convince yourself that the abstract domain is complete – that is, any possible concrete value can be mapped to a type in that domain.)

Program 1

```
def f() {
                [ x -> ____ ]          <- abstract store at this program point
  x = 1;
                [ x -> ____ ]          <- abstract store at this program point
  x = x * 0;
                [ x -> ____ ]          <- abstract store at this program point
  x = 1 / x;
                [ x -> ____ ]          <- abstract store at this program point
  return x;
                [ x -> ____ ]          <- abstract store at this program point
}
```

Questions

- What is the initial abstract store for each program point?
- Would the initial abstract store be different if x was not a local variable but rather a formal parameter of the function x?
- Where did you apply the abstraction function, transfer function(s), or the lub?
- After abstract interpretation, what do you know about the values that f can return?

Program 2

```
def f() {
  if (*) {                             <- condition '*' is independent of x
    x = 1;
  } else {
    x = 0;
  }
  x = 1 / x;
  return x;
}
```

Questions

- How many abstract stores are there; what are their initial mappings for x?
- Where did you apply the abstraction function, transfer function(s), or the lub?
- After abstract interpretation, what do you know about the values that f can return?

Program 3

```
def f() {
  x = 10;
  while (x != 0) {
    x = x -1;
  }
  x = 1 / x;
  return x;
}
```

Recall that this example requires a fix-point analysis: given the initial abstract stores, follow the control flow and update the abstract stores until they no longer change.

Questions

- How many abstract stores are there; what are their initial mappings for x?
- Where did you apply the abstraction function, transfer function(s), or the lub?
- After abstract interpretation, what do you know about the values that f can return?