Program Analysis **October 18, 2015**

Lecture 1: Introduction

Lecturer: Mooly Sagiv *Scribe: Oded Elbaz, Tomer Greenwald*

1.1 Motivation

Software can be found everywhere. PC, electrical equipment, watches, cars and phones are just the tip of the iceberg of this phenomena. The amount of different products that rely on code, and the variety of different use cases grow every day. We expect software to work according to specifications but if it fails to do so it may not do its job, and the outcome, sometimes, may have dire consequences.

Another example for our motivation, comes from the field of information security. Bugs in the code can lead to exploits that allow an attacker to execute malicious code over a remote device.

All of these reasons and many more lead us to develop and evolve ways of validating the correctness of programs.

1.1.1 Example

We will use a simple C implementation for the Bubble Sort algorithm to demonstrate some bugs that we would like to identify:

```
int *array = 0, n, c, d, swap;
int MAX_ARRAY_SIZE = 100;
printf(Enter number of elements\n);
scanf(%d, &n);
array = malloc((n > MAX_ARRAY_SIZE ? MAX_ARRAY_SIZE : n)*sizeof(int));
printf(Enter %d integers\n, n);
for (c = 0; c < n; c++)

    scanf(%d, &array[c]);

for (c = 0 ; c < ( n - 2 ); c++) {

    for (d = 0 ; d < n - c - 1; d++) {
        if (array[d] > array[d+1]) {
            swap       = array[d];
            array[d]   = array[d+1];
            array[d+1] = swap;
        }
    }

}
printf(Sorted list in ascending order:\n);
for ( c = 0 ; c < n ; c++ )
```

```
        printf(%d\n, array[c]);
    free(array)
```

These bugs can be characterized by:

- Out of bounds memory access — The user supplies the amount of numbers it would like to store and sort. Then the program reads each number from the user and stores them in the array. We can notice that the program might access an un-allocated memory when iterating over the array. This happens because we iterate over n cells instead of min(n, MAX_ARRAY_SIZE).
- Loops — The first for loop mistakenly misses the last cell in the array and therefore it will not be sorted.
- Dynamic heap allocation — We must follow allocated memory and associated pointers to know if a dangling pointer is being dereferenced. The free function at the last line is dangerous since we do not check if the memory has been initialized (the user may have entered zero as the number of elements to sort).

1.2 Program Analysis Difficulties

Program analysis is a difficult task. Even a "simple" question as whether a code block is reachable, is undecidable since we can reduct the halting problem to a reachability problem.

We can take a look at Rice's theorem which is a generalization of the halting problem. The Rice Theorem claims that any nontrivial property about the language recognized by a Turing machine is undecidable. A property about Turing machines can be represented as the language of all Turing machines, encoded as strings, that satisfy that property.

The problem of proving nontrivial properties has several difficulties:

- Coding languages are complicated — Variable types can be large (long, float), the system model is unbound, there are unbounded loops, user can define custom data types, code sometimes is missing (if we use external DLLs or shared objects), etc.
- Determining which properties should be tested is not clear.
- Great time complexity — The method we will see has a quadratic time complexity (and sometimes even more), and therefore is not feasible for long source code.

1.3 Abstract Interpretation

Abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g. control-flow, data-flow) without performing the actual calculations.

The main concept is that instead of calculating properties over the *concrete domain* we will transform it into an *abstract domain* (which is a different domain that conforms to certain relations with the *concrete domain* that we will explain later on) and have our calculations done in the *abstract domain.*

This transform and the analysis rules assure us that the calculations are made over elements in the abstract domain that their concretization is a larger (or at least not smaller) group of states than the reachable state diagram in the concrete domain. At the end of the analysis we will concretize the result and it is promised to be an over-approximation of the concrete reachable states group. It insures us soundness (we will find all bugs in the program) but we loose precision as a trade off (we might trigger false alarms).

The rationale behind working with over-approximated groups is that they are mostly easier to prove properties for, and we may also consider states which do not occur at runtime.

The analysis starts with an abstraction of the initial state group. From now on we will have all of our calculations done over the abstract domain. We traverse through the program instructions and at each step, add all reachable states from the current group. This way we over approximate the reachable state diagram - the actual reachable state diagram is a subset of the one we have. E.g. let's assume that we are processing the code and have reached an if-else clause. We will add to the reachable state group the result of two possibilities — states that are reachable from the if-clause execution and states that are reachable from the else-clause execution.

When we will check the result we would not want it to overlap the unwanted states. In this case the code is guaranteed to be safe, otherwise we have found a bug or have raised a false alarm.

The figures 1.1-1.3 illustrate the possible relations between the reachable state, the analysis result and the bad states.

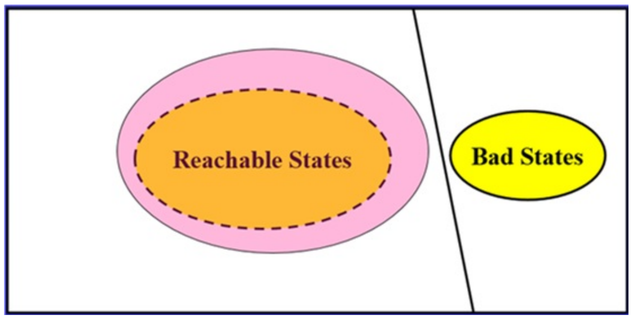


Figure 1.1: Over-approximation — The group of reachable states for the program is over-approximated but still keeps a safe distance from the group of bad states.

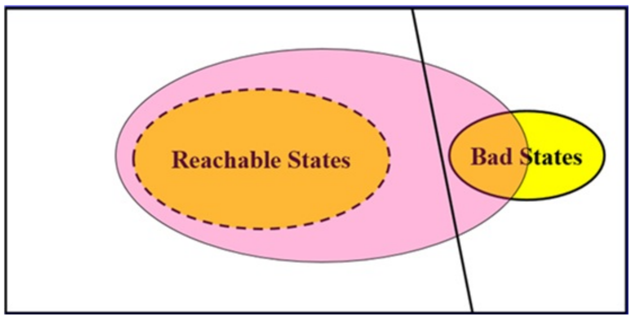


Figure 1.2: False alarm — The group of reachable states for the program is over-approximated, but the approximation is too coarse. This leads us to falsly notify about an error that does not exist.

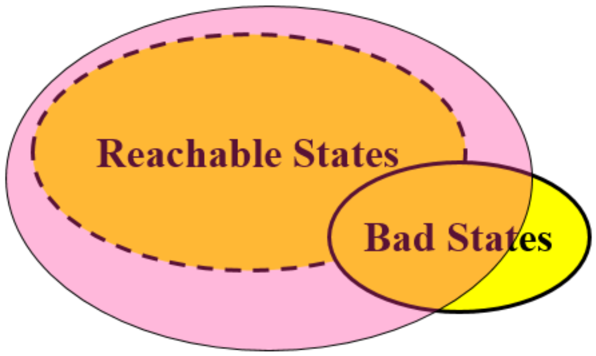
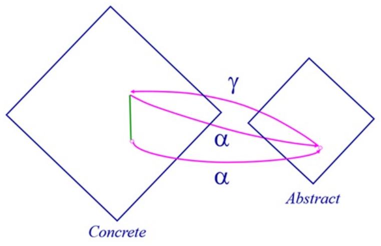


Figure 1.3: Bug — The group of reachable states for the program is over-approximated. This leads us to notify about an error that does exist.

The abstract and concrete domains are connected Galois Connection (invented by variste Galois as part of the Galois Theory) between the concrete and the abstract domain. It will require us to use gamma (concretization from the abstract to the concrete domain) and alpha (abstraction from the concrete to the abstract domain) functions that have the following connection $\gamma(\alpha(s)) \geq s$.

The following illustration shows that concretization of an abstraction of every element from the concrete domain should be equal or larger than the original element. We can see in the illustration two objects from the concrete domain that are mapped to a single object in the abstract domain. When the matching object in the abstract domain is mapped back to the concrete domain it is mapped to a group in the lattice that is larger (or at least not smaller) than the group containing each of the original concrete values.



We should notice that sometimes we will find esoteric bugs that might never be reached in practice but are still reachable by a certain input to the program.

A specific program execution path that leads to a certain state we calculated might not always be visible (i.e. whether an if branch was taken, how many times a loop is executed, whether it stops at all, etc.) since we do not actually execute the program (and are prone to errors such as the amount of times a loop is actually cycled through), but we will know an over-approximation of the possible values of the data at each program state.

1.3.1 Abstract Interpretation Disadvantages

The abstract interpretation we have just learnt has some disadvantages that should be noticed:

1. Analysis efficiency - Programs that implement abstract interpretation are mostly inefficient and take

great memory usage when they are executed. For this reason we have problem using them for analysis of long source code since the time and space complexity are directly related to the length of the input (the source code).

2. Non-Trivial - The properties that we want to prove are not easy to pick.
3. Correctness and Precision - Depends on the abstraction and the analyzed program our analysis will sometimes over-approximate too much and which will cause false alarms, despite the fact that the program holds the tested property.

1.4 Examples

1.4.1 Interval Domain

In interval domain each variable will be associated with one interval that will be noted as [from,to] where *from* and *to* are either an integer, -inf or +inf (for negative and positive infinity accordingly). The abstraction states that the concrete value of the variable must be in the range specified by the abstract value, the interval.

When we do not know which branch took place we use a Join method to take two, or more, possible states and join them into one interval representing the abstract state that follows.

The following examples shows uses of the abstract domain:

```
if (x>0) {
    y:=2
} else {
    y:=x
}
```

At the beginning of the program we do not know anything about the values of x an y, thus we will set both of their intervals to [-inf, inf], which is the Top value (the biggest element in the lattice). We are choosing this value since this is the only way to stay sound, the actual value of x and y must be smaller from the concretization of top (which is also [-inf,inf]).

Prior to the execution of the second line, we know that the condition was evaluated to True (otherwise we would not branch here and instead jump to the fourth line), so we can deduce that x's interval is (0, inf], but we still do not know anything about y (so we do not change its value). After the execution of this line we know that y's interval is [2,2].

Prior to the execution of the fourth line, we know that the if expression evaluated to False (otherwise the else block would take place), so we can deduce that x's interval is [-inf, 0], but we still do not know anything about y. After the execution of this line we know that y's interval is [0, inf].

Since we do not know which of the two branches will take place during run-time we will over-approximate the reachable state diagram and will Join the results of the second and fourth lines. The Join operation must over-approximate the group of reachable state in line 5 in order to stay sound, so we can join the intervals that are reachable from both the *if* and *else* branches. We will conclude that x's range is [-inf, inf] and y's range is [-2, inf]. Even though -1 is not an actual option we must include it in the interval because we must choose only one interval (this is the abstract domain) and this is the only option to do so. It is ok since it is an over-approximation, so we are still sound. Researches show that we are usually not interested in a single value but in a group of them - intervals.

```
x:=2
y:=0
while(True) {
    x:=x+y
    y:=y+1
}
```

Before the loop execution we can use the interval [2,2] for x as an abstract state and [0,0] for y as intervals.

After the first execution of the loop the intervals will be updated to [2,2] and [1,1] for x an y accordingly. After another execution of the loop the intervals will be updated to [3,3] and [2,2].

It is clear that the analysis result does not converge, and since we cannot infinitely execute the loop we will use a trick called widening. Since x grows slower than y during the first execution of the loop we will use the widening trick over y.

We noticed that y grows every iteration of the loop so we will over-approximate it's interval to [0, inf]. After the execution of the fourth line x's intervals will be [2,inf] and after the fifth line will be executed y'2 interval will be [0, inf]. Joining these intervals with the set of states from line 3 will yield [2, inf] and [0, inf] for x and y accordingly.

Now we have reached a convergence since running the loop one more time will not change the variables intervals.

We should notice that the widening process is limited by the amount of variables that are involved in the program execution.

Let's try another example. Assume we have a code for a book library management. We would like to know the amount of books that have been taken since we would like to limit this number by a hundred (as an arbitrary bound).

The code for managing the library is as follows:

```
int taken_books=0;
while (true) {
    char operation=getc()
    if (operation=='t') {
        taken_books++;
    } else {
        taken_books--;
    }
}
```

We will start our analysis with a base interval for *taken_books* - [0,0]. After a single iteration of the loop we will adapt the interval to [-1, 1] because we do not know which of the two branches (if or else) took place. It is clear that at the *i-th* iteration of the loop the interval will be [-i, i]. We wanted to assert that the amount of taken books is at most 100 but from the analysis, and the fact that the matching interval for *taken_books* after the 101-*st* iteration will be [-101, 101], we can conclude the analysis found a bug.

1.4.2 Shape Analysis

Shape analysis has been the focus of the group working at the Tel Aviv University for many years. It is one of many Abstract Implementation applications. It focuses on proving properties regarding dynamically allocated memory.

Properties one may prove with shape analysis:

- Does x point to
  - an acyclic list
  - a tree
  - a directed acyclic graph
- A data structure invariant is correct
- Identify pointers that may point to the same location
- Identify pointers that point to different structures
- Identify null dereferences
- Assert no memory leaks are present

e.g.

```
int *p, *q
q = (int*)malloc()
p=q
*p=5
p=(int*)malloc()
printf(*q)
```

In this example 5 will be printed to screen since q points to a location where it's value is 5. We should notice that despite the fact that at the end of the program p and q point to different location we would still require our analysis to follow memory management and identify memory load and store operations.

The next example shows why working with dynamically allocated memory is difficult:

```
head = null;
while (True) {
    value = rand()
    iterator = head;
    flag = false;
    while (iterator != null) {
        if (iterator.value == value) {
            iterator.n++;
            flag = true;
        }
        iterator = iterator.next()
        }
        if (!flag) {
            t = malloc();
            t.value = value;
            t.n = 1;
            t.next = head;
            head = t;
        }
    }
```

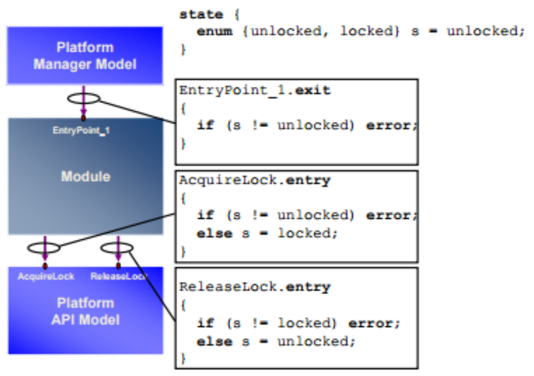
This program generates a linked list with random values, and keeps, for every random value, how many times it was was picked from a random generator. We would like, for example, validate for every variable we randomize -that we are iterating over the entire list (otherwise we may insert an existing value to the list). Another property that we would like to validate is that we don't loose elements, meaning that the linked list contains all the values we have randomized so far. We will fail to represent each memory cell generated along the running course of the program since our analysis program will run out of memory storage at some point. Therefore we would like to find a more compact way to describe memory and connections between variables. Shape analysis will be our savior for this one.

In shaped analysis we prefer to keep information about where pointers point to but our less inclined to remember information that is usually not so relevant to our verification process such as the size of the memory. Hence we will keep every node that we are pointing to (we assume that the number of variables in the code is small enough and therefore the number of nodes we need to track will be of equal proportions). Every other node will be grouped into a summary node. A pointer that points to a summary node may point to every node in that group.

This way we would be able to prove interesting properties for data structures with a runtime that is relatively low and practical. The main issue with this type of analysis is the memory management.

1.4.3 SLAM

SLAM is a program engine developed by Microsoft to verify that an interface between a program and an external code works correctly. Microsoft uses this tools mainly for the verification of device drivers that are written by third party companies. This tool is used to lower the number of errors caused by misusing the API device drivers are supplied with and offer, which usually lead the operating system to crash.



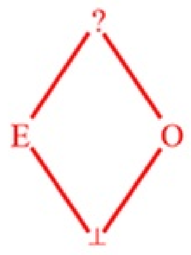
```
state {
  enum {unlocked, locked} s = unlocked;
}

EntryPoint_1.exit
{
  if (s != unlocked) error;
}

AcquireLock.entry
{
  if (s != unlocked) error;
  else s = locked;
}

ReleaseLock.entry
{
  if (s != locked) error;
  else s = unlocked;
}
```

The interaction between the platform model, a module, and a rule. The platform manager model calls into the entry points of the module. The module itself interacts with the underlying API model, while the rule specifies the safe interactions between the various components.

1.4.4 Even/Odd Domain

This type of analysis can be used to determine whether a variable is odd or even at a certain stage of the program execution.

The abstract domain we will use is the following:



Bottom is the smallest abstract value. It is the abstraction of the empty state. The concrete value of it is the empty set as well.

Top is the largest value in the abstract domain. It is the abstraction of states that contain both odd and even values. I.e. at some program state, a variable x, can be either odd or even.

E is the abstract value for states that contain only even values. E.g. if, at a given program state, a variable, x, can be one of the following {2,4,6} in the concrete domain then its matching value in the abstract domain will be E. Likewise O for odd numbers.

The following example shows how we can use this abstract domain:

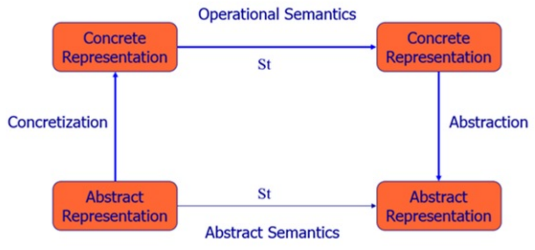
```
while(x!=1) do {
    if (x%2==0)
        {x:=x/2}
    else
        {x:=x*1+1}
}
```

At the beginning of the program we do not know x's value so we will set x=?. This state will still be correct before the execution of the second line. Before executing the third line it is clear the x=E (otherwise the if branch would not be jumped to), but after the execution of the third line x can be either even or odd so it will be x=?. Before the execution of the fifth line it is clear that x=O and after it x=E.

After the execution of the program is complete, since we escaped the loop we know that the concrete value for x is 1 and therefore x=O.

1.5 Best Abstract Transformer

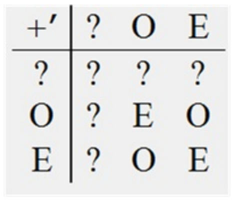
The best abstract transformer is demonstrated by the following chart:



It will be one where executing a statement over an abstract state under the abstract semantics will lead us to a new state that is equal to the one we would get if we were to concretize the initial abstract state, perform the statement using the concrete semantics and then abstract it back to the abstract domain.

1.5.1 Example

Let's assume that our concrete domain is the natural numbers domain (including 0). The semantics for the addition under this domain are the same as we know from math. The transformer from the concrete semantics to the abstract will be best if we will set the addition operation in the abstract domain to be as follows:



+′	?	O	E
?	?	?	?
O	?	E	O
E	?	O	E

In the concrete domain, adding an element from a group containing only even numbers, with another element, also from an only even elements group, will result also an even number, therefore the + relation in the abstract domain must keep E+E=E in order to be a best transformer.

Addition of a group containing both even and odd numbers (?) with a group containing only even numbers (E) will results in a group containing either even or odd numbers (?) and therefore ?+E=?.

Assume, for example, that we had changed the meaning of + in the abstract domain to be E+E=?, O+O=?. If we concretize the state before the action and then abstract it back, we would get E, which is smaller than ?. It means, for this case, that the abstract semantics lost precision, although we are staying sound. We would prefer to work with best transformers because they are more accurate and therefore we will manage to prove more properties.

Another example would be the following code:

```
int n;
n = scanf(%d, &n);
final_n = n+n;
int array[final_n];
for (int i=0; i<final_n; i++)

    array[i] = i;

for (int i=0; i < final_n; i:=i+2)
```

```
        { // doing some action on array[i] and array[i+1]
```

We would like to prove the following property — every element was accessed at least once during the loops body execution.

In the abstract domain under the changed rules , *final_n* will get the value ? and therefore the analysis of the loop will not succeed in proving that we are iterating over each item in the array.

The following example shows how we can use this domain to assert a variable is even at the end of program execution:

```
x=5
y=7
if(getc())

    {y:=x+2}

z:=x+y
assert(z%2=0)
```

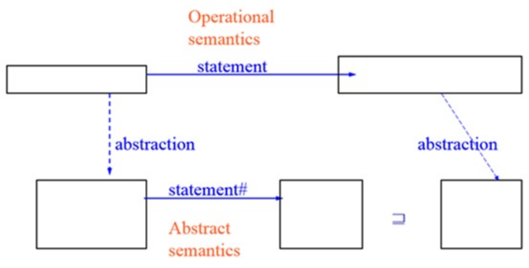
This program demonstrates a state where a random input is given by the user. Since the program execution path relies on that input we are unsure whether the if branch will be taken but we would still like to assert that z is even at the end of the program's execution. Let's analyze:

Statement	Abstract State Before Statement	Abstract State After Statement
	{x=?, y=?, z=?}	{x=?, y=?, z=?}
x=5	{x=?, y=?, z=?}	{x=O, y=?, z=?}
y=7	{x=O, y=?, z=?}	{x = O , y = O, z=?}
if(getc())	{x = O , y = O, z=?}	{x = O , y = O, z=?}
y:=x+2	{x = O , y = O, z=?}	{x = O, y = O, z=?}
z:=x+y	{x = O, y =O, z=?}	{x = O, y = O, z=E}
assert(z%2=0)	{x = O, y = O, z=E}	{x = O, y = O, z=E}

We should notice that x and y's values before the execution of the fifth line is a Join (the same action we saw previously done over intervals) between the after of the fourth line and the before of the third line. We use Join since we do not know which execution path will be picked, as discussed previously — an over-approximation.

1.6 Local Soundness of Abstract Interpretation

The general soundness for the semantics is given by the local soundness of every action in it. The meaning will be that the semantics and the transformer for every possible statement will follow:



Abstraction of the state we get after executing the statement in the concrete semantics must be equal to or less than to first abstract the concrete state and then execute the statement in the abstract domain. This relation is only possible due to the Galois connection. This connection enforces that a group is always a subset or equal to the abstraction of its concretization. For this reason these two groups may not always be homomorphic.

We allow precision loss, but whatever property we prove for the abstract state from executing the statement in the abstract semantics must also be true for the group that we will get by abstracting the state we reached by performing the statement in the concrete semantics.

1.6.1 Example

Let's assume that the concrete domain is the Integer domain and the abstract domain will be P and N which are abstractions for the Positive and Negative possible values accordingly. We will show an example for the local soundness of the addition statement:

We will start by having x=-8, y=7. We would like to execute x:=x+y. Using the concrete semantics will lead us to x=-1, y=7 and the abstraction of this state will be x=N, y=P. On the other hand, an abstraction of the initial state will be x=N, y=P and performing the statement using the abstract semantics will yield x=? (since the concrete value for x might be either positive or negative), y=P. Thus we lost precision for x. But, x=N, y=P is lower (in the lattice) than x=?, y=P and therefore we have kept the local soundness.

1.7 Additional Proving Techniques

There are many more proving techniques that we will not be able to cover during our course:

1. Dynamic Analysis - Performs analysis during the execution of the program. The results of the test, its speed and the statements executed will rely on the input for the program. The time complexity for the analysis will be the same as the time complexity for the program execution.
2. Fuzzing (Miller) - Testing the program by feeding it with random inputs. This method is relatively simple and does not yield false alarms. It is clear that finding bugs might take long time using this technique and sometimes we might also find bugs that are of no interest due to their random nature.
3. Bounded Model Checking - Using this method will require us to use a verifier that will be given as input the program we would like to verify, a number K that will be the upper bound for the number of steps we unwind every loop and a property we would like to check. The verifier will create an equation that will describe the running course of the program for every 1,..,k iterations of the loop. Then it will pass this equation AND the negative of our property to a SAT solver. This model relies on the verified

program to be supplied as an equation. If the SAT will determines that the equation is satisfiable then there is a state where the property does not hold.

4. Deductive verification - Using this method will require the programmer to give the verifier an inductive invariant for every loop in the program. A loop invariant is a property that holds for every execution of the loop. An inductive loop invariant is an invariant that holds regardless of the state the program was holding before entering the loop. E.g. - the inductive invariant guarantees that if a pre-condition holds, the invariant and the pre-condition hold for every loop iteration, regardless the state of the rest of the program. The verifier will create a formula using the invariant that was given by the programmer to represent the program. Similarly to the previous methods will try to use a SAT solver to prove that the property does not hold. Unlike Bounded Model Checking, it might be that the problem with proving the property is not caused by the program itself but by the inductive invariant the programmer supplied. We will learn automatic methods for invariant generation during the course but they will different in nature from invariants that are given by the programmer.