0368.4435 Automatic Software Verification **April 28, 2015**

Lecture 7: Introduction to Abstract Interpretation

Lecturer: Mooly Sagiv, TA: Oded Padon *Scribe: Oded Elbaz*

7.1 Introduction

In the last few lessons we learned several methods for proving correctness of programs. The biggest problem we discussed was finding inductive loop invariants. Usually this task was imposed on the programmer himself. This responsibility turns out to be a complicated task, since finding an inductive loop invariants is difficult. Sometimes it turns out to be even more complicated than writing a perfectly working code.

We will learn today a method that deduces those invariants as the analysis progresses. Although it is sound perfect for our purposes, we will see that it can be costly.

7.2 Static Analysis - Overview

The main advantage of *static analysis* is automatically deducing sound and inductive loop invariants:

- *Sound*: If the analysis passes, i.e; no bugs have been found during the analysis, then the code does not contain bugs. The other way around does not necessarily hold. The analysis may not pass, although the code is bug free.
- *Inductive*: The invariant is implied by the loops precondition and is preserved in each iteration of the loops body. Since the correctness of inductive loop invariants can be checked locally given the loop precondition, inductive invariants play a key role in many program verification systems.

As you can notice, the main disadvantage of this method is **False Alarms**, meaning that the analysis may not pass, but the code does not contain any bug. Although the false alarms are undesirable, it is probably the best we can do, since by reduction from the halting problem it can be proved that for every *Turing complete language* the problem of finding bugs in all the programs that may be written in this language, is undecidable.

In static analysis, as it's name implies, the program is not actually executed during the analysis process. Instead we analyse it statically by creating a system of equations that descries the values in each statement and solve it.

Regarding effectiveness, static analysis can find rare bugs, but it suffers from false alarms. In contrast, runtime analysis methods usually do not alert on false alarms, but can miss bugs. Regarding complexity, static analysis methods are proportionate to the code complexity, while runtime analysis methods are proportionate to the runtime complexity.

Static analysis is highly popular with the developers community. Nowadays it is common that compilers shipped with integrated some static analysis features.

Today we will see some simple examples, and we will learn the general concepts and features of static analysis, while in the next lesson we will dive into the details.

7.3 Examples

In the current section we will see some examples that will help us understand the power of static analysis. The notes in the code are the deduction of the program state that the analysis performs.

Example 1:

```
1  main() {
2    int i = 0, *p=NULL, a[100];
3    for (i=0 ; i <= i < 100)
4    {
5      # { 0 <= i < 100 }
6      a[i] = i;
7      # { p == NULL }
8      p = malloc(1, sizeof(int));
9      # { alloc(p) }
10     *p = i;
11     # { alloc(p) }
12     free(p);
13     # { !alloc(p) }
14     p = NULL;
15     # {p==NULL}
16   }
17 }
```

In this example we are trying to analyze the validation of the code memory accesses. As you can see, the code does not contains bugs; we are freeing only allocated memory, there is not double allocate, etc. Since the analysis is sound, but suffers from false alarms, it is not guaranteed that the analysis will pass. However, in this case we can see that it did.

Example 2:

```
1  main() {
2    int i = 0, *p=NULL, a[100], j;
3    for (i=0; i < j; i++)
4    {
5      # { 0 <= i < j }
6      a[i] = i;
7      p = malloc(1, sizeof(int));
8      # { alloc(p) }
9      p = malloc(1, sizeof(int));
10     # { alloc(p) }
11     free(p);
12     free(p);
13   }
14 }
```

Since the analysis is sound and the code contains bugs (e.g. if j is bigger than 100 we will access a memory outside of the array, memory leak (line 7-9), double free (line 12)), it is guaranteed that the analysis will not pass.

Example 3:

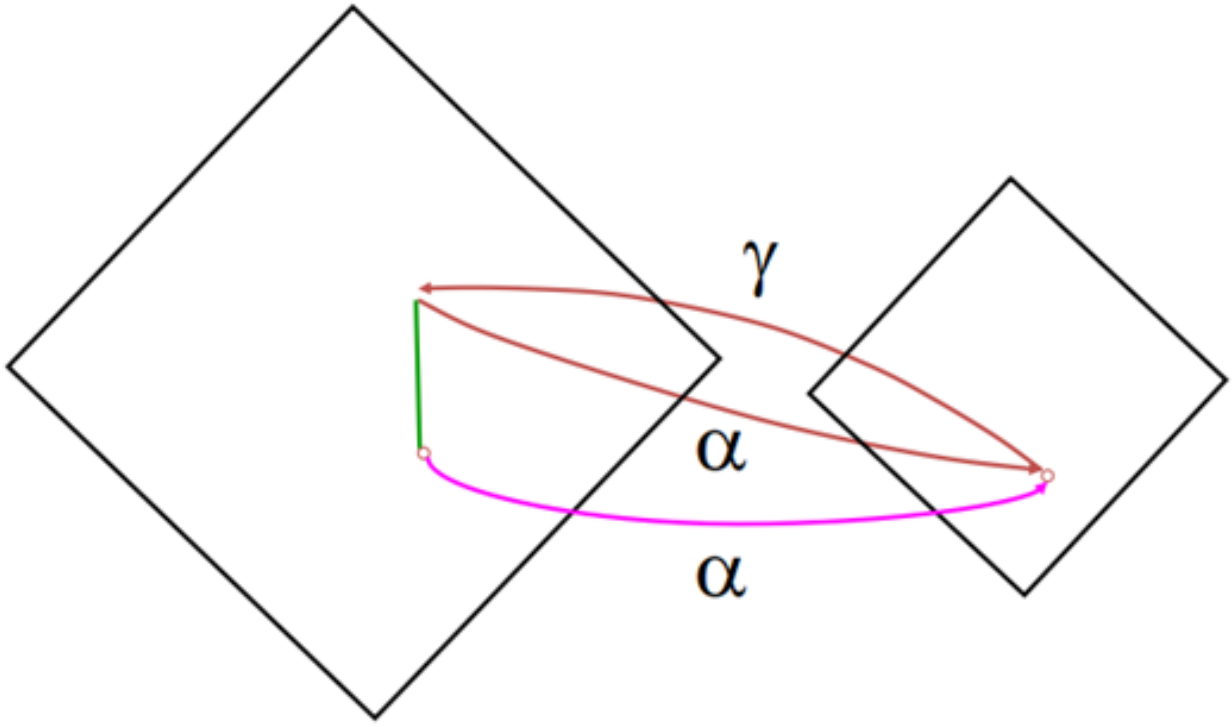
```
1  int i, *p=NULL;
2
3  if (i >=5) {
4    p = malloc(1, sizeof(int));
5  }
```

```
6
7  if (i >=5) {
8    *p = 8;
9  }
10
11 if (i >=5) {
12   free(p);
13 }
```

Although the code does not contain bugs, the analysis will not pass, but raise a false alarm. This happens because the analysis will not manage to deduce that the evaluation of the three *if*s are equal.

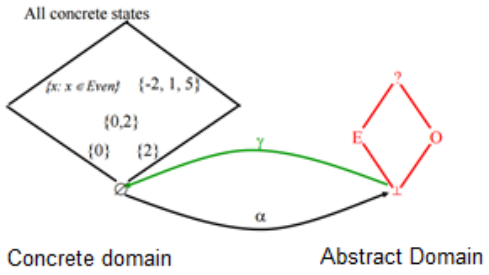
7.4 Static Analysis in a Nutshell

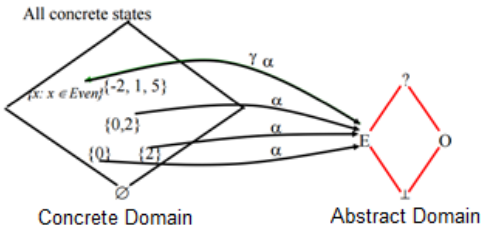
The static analysis is done over an **Abstract Domain**. Every element in this domain can describe more than one element in the **Concrete Domain**. In other words, several elements in the concrete domain can be mapped to a single element in the abstract domain, but there are no two elements in the abstract domain that are mapped to the same element in the concrete domain. An illustration to the relations between the two domains can be seen as:

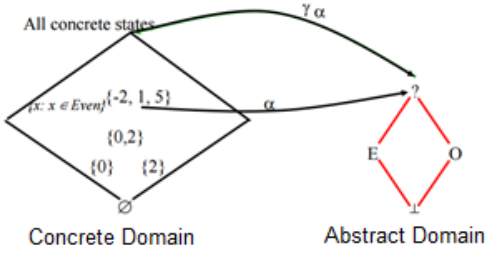


The left rhombus is the concrete domain, the right rhombus is the abstract domain, the α-function maps elements from the concrete domain to the abstract, and the γ-function maps elements from the abstract domain to the concrete. Interesting things in this illustration:

- There are two elements in the concrete domain that mapped to the same element in the abstract domain.
- For c-upper, the biggest element in the concrete domain, c-upper=$\gamma(\alpha$(c-upper)).







- For every element c in the concrete domain, c<=γ(α(c)). This connection is called **Galois connection**.

7.4.1 Example: Even/Odd Abstraction

```
1  /* x=? */
2  while (x != 1) do {    /* x=? */
3    if (x % 2) == 0 {      /* x=E */
4      x := x / 2;    /* x=? */
5    }
6    else {     /* x=O */
7      x := x * 3 + 1;    /* x=E */
8      assert (x % 2 ==0);
9    }
10 }
```

We chose the Even-Odd abstract domain for this code analysis.
At the beginning of the program we do not know what is the value of x, therefore we describe it as ? which represents the set in the concrete domain that contains all of the elements in the domain. If the *if* part is taken, then we know that x is even, otherwise x is odd. We also know the semantics of the arithmetic numbers, therefore we know the result of the multiplication and addition.
Note that we do not know how many iterations will the loop perform, but we know the abstract representation of those values when we perform it.
A figure illustrating the α-function and the γ-function is attached above. The left side of each illustration is the concrete domain. In this domain, the bottom element (the smallest element) is the empty set, the biggest element is the set of every possible integer. Between the biggest element and the smallest one, exist all the partial sets. The right side of each illustration is the abstract domain. It contains a bottom element that is mapped to the empty set (and vice versa), a top element that represents all the sets in the concrete domain that contain both even and odd elements, and two more states, E and O, that represents the sets that contain only even or odd numbers respectively.

7.4.2 Static Analysis Algorithm

First Create a system of equations over the abstract domain.

Second Iteratively compute the **smallest** solution for the system.

7.4.3 Example: Interval analysis

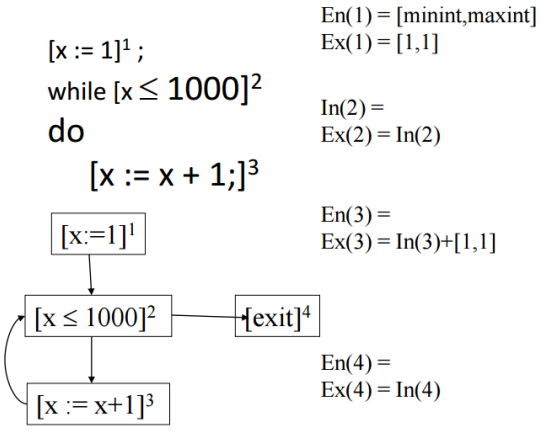
The interval domain is one of the most common abstract domains. We will use it to analyze this program:

```
1 x := 1;
2 while(x <= 1000)
3 do
4   x := x + 1;
```

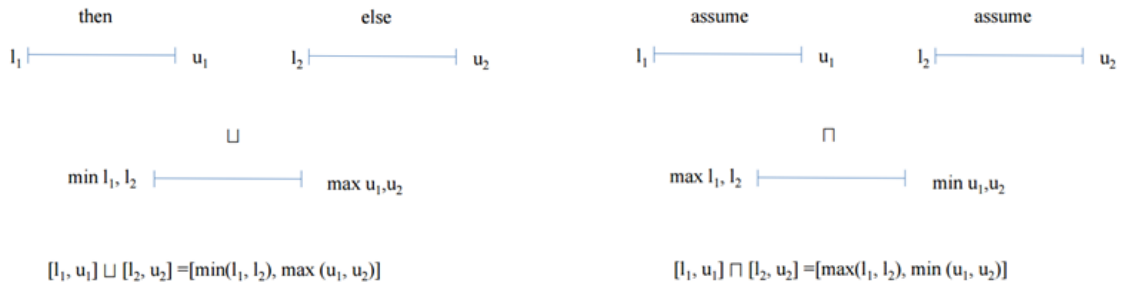
The functions over the interval are defined as:

- ⟦skip⟧[1, u] = [1, u]
- ⟦x:=1⟧[1, u] = [1, 1]
- ⟦x:=x+1⟧[1, u] = [1, u] + [1, 1] = [1+1, 1 + u]

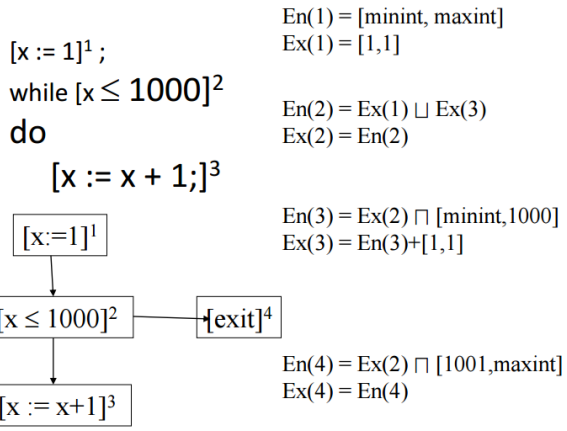
We will start with describing the entry value (En) and exit value (Ex) for every statement in the program:



To understand the values of En(2), En(3), En(4), we must define two new operations: join and meet. The figure attached above defines (informally) those operations.
Note that statement 2 has two entry points - the exit of the first and third statements. Therefore the entry of this statement is join between them, i.e. En(2)=Ex(1) ⊔ Ex(3).
Statement 3 occurs after the second statement, and just when the *if* condition does not hold, therefore En(3)=Ex(2) ⊓ [minint, 1000] describes the valid entry interval for this statement.
Statement 4 is similar, En(4)=Ex(2) ⊓ [1001, maxint].
If the meet between the intervals is empty, it means that we found dead code.



Summarizing the system of equations we have found:



This system is iterative, one equation defines another. We will see next lesson that a solution always exists, however, it may be very hard to calculate it. A solution to this equation may be:

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	En[4]	Ex[4]
[-inf, inf]	[1, 1]	[-inf, 1001]	[-inf, 1001]	[-inf, 1000]	[-inf, 1001]	[1001, 1001]	[1001, 1001]

Remember, however, that we are interested in the least solution:

First Initiate interval En(1) by the program semantics

Second Initiate other intervals as bottom

Third Calculate the solution until there is no change

Notes:

- We can solve the system of equations in any order, it does not influence the result (it may influence the complexity).
- Although we begin with unsound solution we end with a sound one.

7.4.4 Widening and Narrowing

Finding the solution to the system of equations may take a while (as in the example, in every iteration only a small interval is added). In order to speed up the analysis, we assist a function called **Widening**. We use this function to predict the general convergence of the loop. In the common widening function we compare the result of the previous and the current iterations and set the value accordingly.
After executing widening, we usually get a very big result that does not indicate the true result. Therefore after using widening we will use **Narrowing** to amplify the result precision.
It is highly recommended not to use these functions (or use with extra caution), because they cause a loss in precision. In the example we saw in class we use those functions only in the entrance.

7.5 Summary

Static Analysis is well learned and popular method among the developers community. The secret of it's success is the "out of the box" sound solution without demanding any extra effort from the programmer.
During the analysis we produce a system of equations and we solve it iteratively. It is guaranteed that the system has a unique small solution.
The disadvantage of this method is the false alarms it generates. Many researchers are still trying to reduce their count.