Static Analysis: Overview, Syntactic Analysis and Abstract Interpretation

TDDC90: Software Security

Ahmed Rezine

IDA, Linköpings Universitet

Hösttermin 2023

Outline

Overview

Syntactic Analysis

Abstract Interpretation

Outline

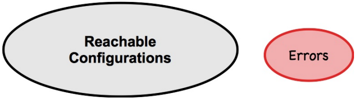
Static Program Analysis

Static Program Analysis analyses computer programs **statically**, i.e., without executing them (as opposed to *dynamic analysis* that does execute the programs wrt. some specific input):

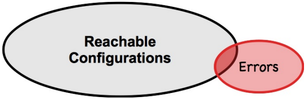
- No need to run programs, before deployment
- No need to restrict to a single input as for testing
- Useful in compiler optimization, program analysis, finding security vulnerabilities and verification
- Often performed on (models of) source code, sometimes on object code
- Usually highly automated though with the possibility of some user interaction
- From scalable bug hunting tools without guarantees to heavy weight verification frameworks for safety critical systems

Program verification and Approximations

We often want to answer whether a program is **safe** or not (i.e., has some erroneous reachable configurations or not):



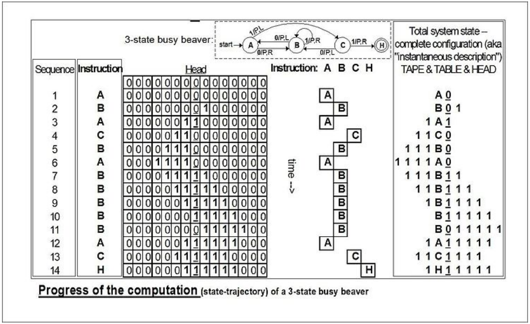
Safe Program



Unsafe Program

The general verification problem is "very difficult"

- Deciding whether all possible executions of the program are error-free is so hard that if we can write an analyzer-program that can always check it for arbitrary programs-to-be-analyzed then we cam always answer whether a Turing machine halts.



- This problem is proven to be undecidable in general, i.e., there is no algorithm that is guaranteed to terminate and to give an exact answer to the problem.

Problem is “very difficult”: what to do?

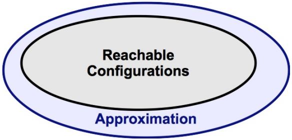
- Identify sub-problems on which one can decide: e.g. finite state machines, push-down automata, timed automata, Petri nets, well-structured transition systems.
- Proceed with approximations that will hopefully give some guarantees.

Static analysis and approximations

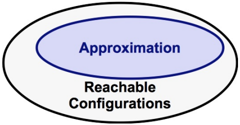
- An analysis procedure takes as input a program to be checked against a property. The procedure is an analysis algorithm if it is guaranteed to terminate.
- An analysis algorithm is **sound** in the case where each time it reports the program is safe wrt. some errors, then the original program is indeed safe wrt. those errors (pessimistic analysis)
- An algorithm is **complete** in the case where each time it is given a program that is safe wrt. some errors, then it does report it to be safe wrt. those errors (optimistic analysis)
- In general, you have to give up on one of the three: termination, soundness or completeness.

Verification problem and approximations

- The idea is then to come up with efficient approximations to give correct answers in as many cases as possible.



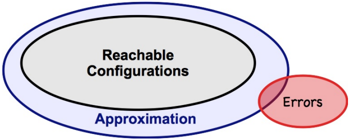
Over-approximation



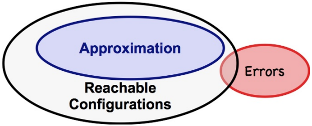
Under-approximation

Program verification and the price of approximations

- A sound analysis cannot give **false negatives**
- A complete analysis cannot give **false positives**



False Positive



False Negative

These Two Lectures

These two lectures on static program analysis will briefly introduce different types of analysis:

- This lecture:
  - syntactic analysis: scalable but neither sound nor complete
  - abstract interpretation sound but not complete
- Next lecture:
  - symbolic executions: complete but not sound
  - inductive methods: may require heavy human interaction in proving the program correct

These Two Lectures

These two lectures on static program analysis will briefly introduce different types of analysis:

- This lecture:
  - syntactic analysis: scalable but neither sound nor complete
  - abstract interpretation sound but not complete
- Next lecture:
  - symbolic executions: complete but not sound
  - inductive methods: may require heavy human interaction in proving the program correct
- These two lectures are only appetizers:
  - There is a deeper course with more tools and applications in the spring (TDDE34)
  - Possibilities of exjobbs with applications to verification.

Administrative Aspects:

- Lab sessions might not be enough and you might have to work outside these sessions
- You will need to write down your answers to each question on a draft.
- You will need to demonstrate (individually) your answers in a lab session on a computer to me.
- Once you get the green light, you can write your report in a pdf form and send it (in pairs) to the person you got the green light from.
- You will get questions in the final exam about these two lectures.

Outline

Automatic Unsound and Incomplete Analysis

- Tools such as the open source *Splint* or the commercial *Clockworck* and *Coverity* trade guarantees for scalability
- Not all reported errors are actual errors (false positives) and even if the program reports no errors there might still be uncovered errors (false negatives)
- A user needs therefore to carefully check each reported error, and to be aware that there might be more uncovered errors

Unsound and Incomplete analysis: Splint

- Some tools are augmented versions of grep and look for occurrences of memcpy, pointer dereferences ...
- The open source Splint tool checks C code for security vulnerabilities and programming errors.
- Splint does parse the source code and looks for certain patterns such as:
  - unused method parameters
  - loop tests that are not modified by the loop,
  - variables used before definitions,
  - null pointer dereference
  - overwriting allocated structures
  - and many more ...

Unsound and Incomplete analysis: Splint

```
...
 return *s; // warning about dereference of possibly null pointer
...
if(s!=NULL)
  return *s; //does not give warnings because s was checked
```

```
int dumbfunc ()
{
  int i;
  if (i = 0) return 1;
  int j=i;
  while(i > 0){
    j--;
  }
  return 0;
}
```

Unsound and Incomplete analysis: Splint

- Still, the number of false positives remains very important, which may diminish the attention of the user since splint looks for “dangerous” patterns
- An important number of flags can be used to enable, inhibit or organize the kind of errors Splint should look for
- Splint gives the possibility to the user to annotate the source code in order to eliminate warnings
- Real errors can be made quite with annotations. In fact real errors will remain unnoticed with or without annotations

Outline

Abstract Interpretation

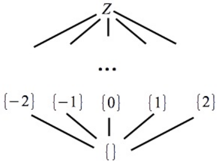
- Suppose you have a program analysis that captures the program behavior but that is too inefficient to be feasible in practice (e.g. enumerating all possible values at each program location)
- You want an analysis that is efficient but that can also over-approximate all behaviors of the program (e.g. tracking only key properties of the values)

The sign example

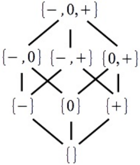
- Consider a language where you can multiply ($\times$), sum ($+$) and substract ($-$) integer variables.
- If you are only interested in the signs of the variables values, then you can associate, at each position of the program, a subset of $\{+, 0, -\}$, instead of a subset of $\mathbb{Z}$, to each variable
- For an integer variable, the set of concrete values at a location is in $\mathcal{P}(\mathbb{Z})$. Concrete sets are ordered with the subset relation $\sqsubseteq_c$ on $\mathcal{P}(\mathbb{Z})$. We can associate $\mathbb{Z}$ to each variable in each location, but that is not precise. We write $S_1 \sqsubseteq_c S_2$ to mean that S_1 is more precise than S_2.
- We approximate concrete values with an element in $\mathcal{P}(\{-, 0, +\})$. For instance, $\{0, +\}$ means the variable is larger or equal than zero. For A_1, A_2 in $\mathcal{P}(\{-, 0, +\})$, we write $A_1 \sqsubseteq_a A_2$ to mean that A_1 is more precise than A_2.

The sign example: concrete and abstract lattices

- A pair $(Q, \preceq)$ is a lattice if each pair p, q in Q has
  - a greatest lower bound $p \sqcap q$ wrt. $\preceq$ (aka meet), and
  - a least upper bound $p \sqcup q$ wrt. $\preceq$ (aka join)
- $(\mathcal{P}(\mathbb{Z}), \sqsubseteq_c)$ and $(\mathcal{P}(\{-, 0, +\}), \sqsubseteq_a)$ are lattices



Concrete lattice
$(\mathcal{P}(\mathbb{Z}), \sqsubseteq_c)$

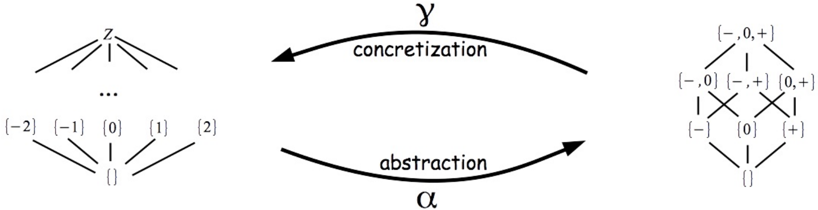


Abstract lattice
$(\mathcal{P}(\{-, 0, +\}), \sqsubseteq_a)$

- For any $S \in \mathcal{P}(\mathbb{Z})$, $\{\} \sqsubseteq_c S$
- If $A_1 = \{-, 0\}$ and $A_2 = \{0, +\}$, then $A_1 \sqcap_a A_2 = \{0\}$ and $A_1 \sqcup_a A_2 = \{-, 0, +\}$

The sign example: Galois connections

- (α, γ) is a Galois connection if, for all $S \in \mathcal{P}(\mathbb{Z})$ and $A \in \mathcal{P}(\{-, 0, +\})$, $\alpha(S) \sqsubseteq_a A$ iff $S \sqsubseteq_c \gamma(A)$
- E.g. here, $\alpha(S) = \{+\}$ if non-empty $S \subseteq \{i | i > 0\}$ and $\gamma(A) = \{i | i \leq 0\}$ if A is $\{-, 0\}$
- Interestingly: $S \sqsubseteq_c \gamma \circ \alpha(S)$ and $\alpha \circ \gamma(A) \sqsubseteq_a A$ for any concrete and abstract elements S, A.



Concrete lattice A Galois connection Abstract lattice

Sound approximations: $f(S) \sqsubseteq_c \gamma \circ g \circ \alpha(S)$

Let A, B be two abstract elements.

$\otimes$	-	0	+
-	{+}	{0}	{-}
0	{0}	{0}	{0}
+	{-}	{0}	{+}

$$A \otimes B = \bigcup_{a \in A, b \in B} a \otimes b$$

$\oplus$	-	0	+
-	{-}	{-}	{-,0,+}
0	{-}	{0}	{+}
+	{-,0,+}	{+}	{+}

$$A \oplus B = \bigcup_{a \in A, b \in B} a \oplus b$$

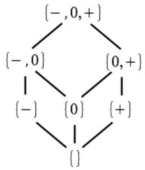
	-	0	+
$\underline{++}$	{-,0}	{+}	{+}

$$A\underline{++} = \cup_{a \in A}\, a\underline{++}$$

	-	0	+
$\underline{--}$	{-}	{-}	{0,+}

$$A\underline{--} = \cup_{a \in A}\, a\underline{--}$$

Example 1



$A \sqsubseteq B$ iff $A \subseteq B$
$A \sqcup B$ is $A \cup B$
$A \sqcap B$ is $A \cap B$
For variable x, assume expression e is captured by $[\![e]\!]$
E.g., $[\![x+1]\!] = \{-,0\}$ if $[\![x]\!] = \{-\}$

$$next_i = curr_i \sqcup \bigsqcup_j (\mathtt{img}(st_{j\rightarrow i}, curr_j))$$

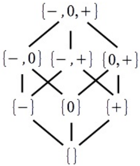
$$\mathtt{img}(x := expr, x : A) = x : [\![expr]\!]$$
$$\mathtt{img}(\mathtt{if}_{\mathtt{then}}(expr), x : A) = x : A \sqcap [\![expr]\!]$$
$$\mathtt{img}(\mathtt{if}_{\mathtt{else}}(expr), x : A) = x : A \sqcup (\top \setminus [\![expr]\!])$$

```
// x: ⊤
while(x>0){
// x: ⊥
  if(x>0){
// x: ⊥
    x--;
// x: ⊥
  }else{
// x: ⊥
    x++;
// x: ⊥
  }
// x: ⊥
  assert(x>=0);
// x: ⊥
}
.
```

```
// x: {-,0,+}
while(x > 0){
// x:  {+}
  if(x > 0){
// x: {+}
    x--;
// x: {0,+}
  }else{
// x: {}
    x++;
// x: {}
  }
// x: {0,+}
  assert(x >= 0);
// x: {0,+}
}
// x: {-,0}
```

```
// x: {-,0,+}
while(x > 0){
// x:  {+}
  if(x > 0){
// x: {+}
    x--;
// x: {0,+}
  }else{
// x: {}
    x++;
// x: {}
  }
// x: {0,+}
  assert(x >= 0);
// x: {0,+}
}
// x: {-,0}
```

Example 2: more precise abstract domain

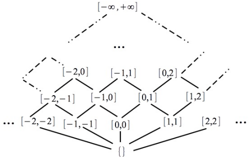



```
//x:⊤, y:⊤
while(x!=0){
//x:⊥, y:⊥
  assert(x!=0);
//x:⊥, y:⊥
  if(x>0){
//x:⊥, y:⊥
    x,y=x--,1;
//x:⊥, y:⊥
  }else{
//x:⊥, y:⊥
    x,y=x++,-1;
//x:⊥, y:⊥
  }
//x:⊥, y:⊥
  assert(y!=0);
//x:⊥, y:⊥
}
//x:⊥, y:⊥
```

```
// x: {-,0,+}; y: {-,0,+}
while(x != 0){
// x:  {-,+}; y: {-,0,+}
  assert(x!=0);
// x: {-,+}; y: {-,0,+}
  if(x > 0){
// x: {+}; y: {-,0,+}
    x,y=x--,1;
// x: {0,+}; y: {+}
  }else{
// x: {+}; y: {-,0,+}
    x,y=x++,-1;
// x:{+}; y: {-}
  }
// x: {0,+}; y: {-,+}
  assert(y!=0);
// x: {0,+}; y: {-,+}
}
// x: {0}; y: {-,0,+}
```

```
// x: {-,0,+}; y: {-,0,+}
while(x != 0){
// x:  {-,+}; y: {-,0,+}
  assert(x!=0);
// x: {-,+}; y: {-,0,+}
  if(x > 0){
// x: {+}; y: {-,0,+}
    x,y=x--,1;
// x: {0,+}; y: {+}
  }else{
// x: {-}; y: {-,0,+}
    x,y=x++,-1;
// x:{-,0}; y: {-}
  }
// x: {-,0,+}; y: {-,+}
  assert(y!=0);
// x: {-,0,+}; y: {-,+}
}
// x: {0}; y: {-,0,+}
```

Example 4: interval domain



$[a, b] \sqsubseteq [c, d]$ iff $c \leq a$ and $b \leq d$
$[a, b] \sqcup [c, d]$ is $[inf\{a, c\}, sup\{b, d\}]$
$[a, b] \sqcap [c, d]$ is $[sup\{a, c\}, inf\{b, d\}]$

$\text{next} = \text{curr} \sqcup \bigsqcup_i (\mathtt{img}(st_i, curr_i))$

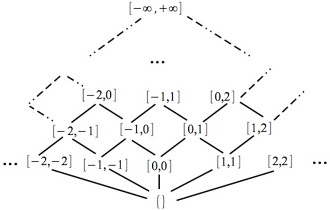
$\mathtt{img}(x := a, x : [l, u]) = x : [a, a]$
$\mathtt{img}(\mathtt{if}_{\mathtt{then}}(x : [a, b]), x : [l, u]) = x : [a, b] \sqcap [l, u]$
$\mathtt{img}(\mathtt{if}_{\mathtt{else}}(x : [a, b]), x : [l, u]) = x : ([-\infty, a-1] \sqcap [l, u]) \sqcup ([b+1, +\infty] \sqcap [l, u])$

```
// x:⊤, y:⊤
x,y=0,0;
// x:⊥, y:⊥
while(x < 100){
// x:⊥, y:⊥
    x,y=x++,y++;
// x:⊥, y:⊥
}
// x:⊥, y:⊥
assert(x>=100);
// x:⊥, y:⊥
assert(y>=100);
// x:⊥, y:⊥
```

```
// x:[-oo,+oo], y:[-oo,+oo]
x,y=0,0;
// x:[0,1], y:[0,1]
while(x < 100){
    // x:[0,0], y:[0,0]
    x,y=x++,y++;
    // x:[1,1], y:[1,1]
}
// x:⊥, y:⊥
assert(x>=100);
// x:⊥, y:⊥
assert(y>=100);
// x:⊥, y:⊥
```

```
// x:[-oo,+oo], y:[-oo,+oo]
x,y=0,0;
// x:[0,2], y:[0,2]
while(x < 100){
    // x:[0,1], y:[0,1]
    x,y=x++,y++;
    // x:[1,2], y:[1,2]
}
// x:⊥, y:⊥
assert(x>=100);
// x:⊥, y:⊥
assert(y>=100);
// x:⊥, y:⊥
```

Example 4: interval domain, widening



[0, 0], [0, 1], [0, 2], [0, 3]....
would take 100 steps to converge.
Sometimes too many steps.
For this use some widening operator ∇.
Intuitively, an acceleration that ensures termination

```
//x:[-oo,+oo],y:[-oo,+oo]
x,y=0,0;
// x:[0,0], y:[0,0]
while(x < 100){
    // x:[0,0], y:[0,0]
    x,y=x++,y++;
    // x:[1,1], y:[1,1]
}
// x:⊥, y:⊥
assert(x>=100);
// x:⊥, y:⊥
assert(y>=100);
// x:⊥, y:⊥
```

```
// x:[-oo,+oo], y:[-oo,+oo]
x,y=0,0;
// x:[0,+oo], y:[0,+oo]
while(x < 100){
    // x:[0,+oo], y:[0,+oo]
    x,y=x++,y++;
    // x:[1,+oo], y:[1,+oo]
}
// x:[100,+oo], y:[0,+oo]
assert(x>=100);
// x:[100,+oo], y:[0,+oo]
assert(y>=100);
// x:[100,+oo], y:[0,+oo]
```

```
// x:[-oo,+oo], y:[-oo,+oo]
x,y=0,0;
// x:[0,+oo], y:[0,+oo]
while(x < 100){
    // x:[0,+oo], y:[0,+oo]
    x,y=x++,y++;
    // x:[1,+oo], y:[1,+oo]
}
// x:[100,+oo], y:[0,+oo]
assert(x>=100);//Ok!
// x:[100,+oo], y:[0,+oo]
assert(y>=100);//False positive!
// x:[100,+oo], y:[0,+oo]
```

Several tools are built on extensions of these ideas, for instance:

`https://antoinemine.github.io/Apron/doc/`