Static Analysis: Overview, Syntactic Analysis and Abstract Interpretation
TDDC90: Software Security

Ahmed Rezine
IDA, Linköpings Universitet

Hösttermin 2014
Outline

Overview

Syntactic Analysis

Abstract Interpretation
Outline

Overview

Syntactic Analysis

Abstract Interpretation
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 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
We want to answer whether the program is safe or not (i.e., has some erroneous reachable configurations or not):
Finding all configurations or behaviours (and hence errors) of arbitrary computer programs can be easily reduced to the halting problem of a Turing machine.

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

An 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.

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.
The idea is then to come up with efficient approximations and algorithms to give correct answers in as many cases as possible.
A sound analysis cannot give **false negatives**

A complete analysis cannot give **false positives**
These two lectures on static program analysis will briefly introduce different types of analysis:

- **This lecture:**
  - syntactic analysis: scalable but neither sound nor complete
  - data flow analysis and 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
Administrative Aspects:

- There will be two lab sessions
- These might not be enough and you might have to work more
- You will need to write down your answers to each question on a draft.
- You will need to demonstrate (individually) your answers in one of the lab sessions on a computer for me (for group A) or Ulf (group B).
- Once you get the green light, you can write your report in a pdf form and send it (in pairs) to me or Ulf.
- You will get questions in the final exam about these two lectures.
Overview

Syntactic Analysis

Abstract Interpretation
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
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
- over writing allocated structures
- and many more ...
Unsound and Incomplete analysis: Splint

Pointer dereference

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

Undefined variables:

```c
extern int val (int *x);

int dumbfunc (int *x, int i)
{
    if (i > 0) return *x; // Value *x used before definition
    else return val (x); // Passed storage x not completely defined
}
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

Overview

Syntactic Analysis

Abstract Interpretation
Suppose you have a program analysis that captures the program behavior but that is inefficient or uncomputable (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).
Consider a language where you can multiply (\(\times\)), sum (\(\,+\)) and subtract (\(-\)) 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 \(\subseteq_c\) on \(\mathcal{P}(\mathbb{Z})\). We can associate \(\mathbb{Z}\) to each variable in each location, but that is not precise. We write \(S_1 \subseteq_c S_2\) 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 \(A_1 \subseteq_a A_2\) to mean that \(A_1\) is more precise than \(A_2\).
The sign example: concrete and abstract lattices

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

\[
\begin{array}{c}
\mathbb{Z} \\
\vdots \\
\{2\} \\
\{1\} \\
\{0\} \\
\{-1\} \\
\{-2\} \\
\emptyset
\end{array}
\]

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

\[
\begin{array}{c}
[-, 0, +] \\
[-, 0] \\
[-, +] \\
[0, +] \\
[0] \\
[+] \\
\emptyset
\end{array}
\]

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

- For any \(S \in \mathcal{P}(\mathbb{Z}), \emptyset \subseteq_c S\)
- 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

- $(\alpha, \gamma)$ 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 $S \sqsubseteq_b \gamma(A)$

- E.g. here, $\alpha(S) = \{+\}$ if $S \subseteq \{i | i > 0\}$ 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
\[ \begin{array}{c}
\{-2\} \\
\{-1\} \\
\{0\} \\
\{1\} \\
\{2\} \\
\end{array} \]
```
The sign example: abstract transformers

Let $A, B$ be two abstract elements.

\[
\begin{array}{c|ccc}
\otimes & - & 0 & + \\
\hline
- & \{+\} & \{0\} & \{-\} \\
0 & \{0\} & \{0\} & \{0\} \\
+ & \{-\} & \{0\} & \{+\} \\
\end{array}
\]

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

\[
\begin{array}{c|ccc}
\oplus & - & 0 & + \\
\hline
- & \{-\} & \{-\} & \{-,0,+\} \\
0 & \{-\} & \{0\} & \{+\} \\
+ & \{-,0,+\} & \{+\} & \{+\} \\
\end{array}
\]

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

\[
\begin{array}{c|ccc}
\underline{\underline{\oplus}} & - & 0 & + \\
\hline
\underline{\underline{\underline{\underline{\oplus}}}} & \{-,0\} & \{+\} & \{+\} \\
\end{array}
\]

$A\underline{\underline{\underline{\underline{\oplus}}}} = \bigcup_{a \in A} a\underline{\underline{\underline{\underline{\oplus}}}}$

\[
\begin{array}{c|ccc}
\underline{\underline{-}} & - & 0 & + \\
\hline
\underline{\underline{-}} & \{\} & \{-\} & \{0,+\} \\
\end{array}
\]

$A\underline{\underline{-}} = \bigcup_{a \in A} a\underline{\underline{-}}$
Example 1

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

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

```
while (x > 0) {
    if (x > 0) {
        x --;
    } else {
        x ++;
    }
    assert (x >= 0);
}
```
Example 2

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

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

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

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

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

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

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

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

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

```
while (x != 0){
    assert (x != 0);
    if (x > 0){
        x, y = x - - , 1;
    } else {
        x, y = x++ , -1;
    }
    assert (y != 0);
}
```
Example 3: more precise abstract domain

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

```
// x: { -,0, +}; y: { -0, +}
while (x != 0)
    assert (x != 0);
if (x > 0)
    x, y = x --, 1;
else
    x, y = x ++, -1;
assert (y != 0);
```
Example 4: interval domain

\[ [a, b] \subseteq [c, d] \iff c \leq a \text{ and } b \leq d \]
\[ [a, b] \cup [c, d] \text{ is } \{ \inf \{a, c\}, \sup \{b, d\}\} \]
\[ [a, b] \cap [c, d] \text{ is } \{ \sup \{a, c\}, \inf \{b, d\}\} \]

```c
x, y = 0, 0;
while (x != 100) {
    x, y = x++, y++;
}
assert (x == 100);
assert (y == 100);
```

```c
// x: [-oo, +oo], y: [-oo, +oo]
x, y = 0, 0;
// x: [0, 0], y: [0, 0]
while (x != 100) {
    x, y = x++, y++;
    // x: [0, 1], y: [0, 1]
}
// x: [], y: []
assert (x == 100);
assert (y == 100);
```
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 \( \nabla \). Intuitively, an acceleration that ensures termination

```c
// x: [-\infty, +\infty], y: [-\infty, +\infty]
x, y = 0, 0;
while (x != 100){
    x, y = x++, y++;
}
assert (x == 100);
assert (y == 100);
```

You can play with more numerical domains at this web interface

http://pop-art.inrialpes.fr/interproc/interprocweb.cgi