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

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

A sound analysis cannot give **false negatives**.

A complete analysis cannot give **false positives**.

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.

Outline

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

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

Pointer dereference

```c
... 
return *s; // warning about dereference of possibly null pointer s 
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
}
```

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

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 $\subseteq_c$. 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 $S_1$ is more precise than $S_2$.

We approximate concrete values with an element in $\mathcal{P}(\mathbb{f}; 0; +)$.

A pair $(Q, \leq)$ is a lattice if each pair $p, q$ in $Q$ has:

- a greatest lower bound $p \cap q$ wrt. $\leq$ (aka meet), and
- a least upper bound $p \cup q$ wrt. $\leq$ (aka join)

$(\mathcal{P}(\mathbb{Z}), \subseteq_c)$ and $(\mathcal{P}(\{-, 0, +\}), \subseteq_a)$ are lattices

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

If $A_1 = \{-, 0\}$ and $A_2 = \{0, +\}$, then $A_1 \cap_a A_2 = \{0\}$ and $A_1 \cup_a A_2 = \{-, 0, +\}$

Let $A, B$ be two abstract elements.

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

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

$$A++ = \bigcup_{a \in A} a++$$

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

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

Example 2

```c
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

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

Example 4: interval domain

```c
[a, b] \subseteq [c, d]
[a, b] \cup [c, d] = [\inf a, \sup b, \sup c, \inf d]
```

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

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

```c
if c \leq a and b \leq d
[a, b] \subseteq [c, d]
[a, b] \cup [c, d] = [\inf a, \sup b, \sup c, \inf d]
```
Example 4: interval domain, widening

\[ [0, 0], [0, 1], [0, 2], [0, 3], \ldots \]

would take 100 steps to converge.

Sometimes too many steps.
For this use some widening operator \( \nabla \).

Intuitively, an acceleration that ensures termination

```plaintext
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