Static Program Analysis
Lecture 1: Introduction to Program Analysis
Winter Semester 2016/17
Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University
https://moves.rwth-aachen.de/teaching/ws-1617/spa/
Preliminaries

People

- Lectures:
  - Thomas Noll (noll@cs.rwth-aachen.de)

- Exercise classes:
  - Christina Jansen (christina.jansen@cs.rwth-aachen.de)
  - Christoph Matheja (matheja@cs.rwth-aachen.de)

- Student assistant:
  - Louis Wachtmeister
Target Audience

- **MSc Informatik:**
  - Theoretische Informatik

- **MSc Software Systems Engineering:**
  - Theoretical Foundations of SSE
Preliminaries

Expectations

- What you can expect:
  - foundations of static analysis of computer software
  - implementation and tool support
  - applications in, e.g., program optimisation and software validation

- What we expect: basic knowledge in
  - programming
    - essential concepts of imperative and object-oriented programming languages
    - elementary programming techniques, ...
  - formal languages and automata theory
    - regular and context-free languages
    - finite and pushdown automata, ...
  - helpful but not mandatory:
    - theory of programming (semantics of programming languages, software verification, ...)
    - implementation of programming languages (compiler construction, ...)

5 of 21 Static Program Analysis
Winter Semester 2016/17
Lecture 1: Introduction to Program Analysis
Preliminaries

Organization

- **Schedule:**
  - Lecture Tue 10:15–11:45 AH 1 (starting October 18)
  - Lecture Thu 10:15–11:45 AH 6 (starting October 20)
  - Exercise class Wed 12:00–13:30 AH 3 (starting October 26)
  - see overview at [https://moves.rwth-aachen.de/teaching/ws-1617/spa/](https://moves.rwth-aachen.de/teaching/ws-1617/spa/)

- **1st assignment sheet** this Thursday, submitted & presented October 26

- Work on assignments in **groups of two**

- **Oral/written exam** (6 credits) depending on number of participants

- **Admission** requires at least 50% of the points in the exercises

- Written material in **English**, lecture and exercise classes “on demand”, rest up to you
Introduction

What Is It All About?

Static (Program) Analysis

Static analysis is a general method for automated reasoning on artefacts such as requirements, design models, and programs.

Distinguishing features:

- **Static**: based on source code, not on (dynamic) execution
  (in contrast to testing, profiling, or run-time verification)
- **Automated**: “push-button” technology, i.e., little user intervention
  (in contrast to theorem-proving approaches)

(Main) Applications:

- **Optimizing compilers**: exploit program properties to improve runtime or memory efficiency of generated code (dead code elimination, constant propagation,...)
- **Software validation**: verify program correctness (bytecode verification, shape analysis, ...)
Dream of Static Program Analysis

Program

Analyzer

Result

Property specification
Introduction

Fundamental Limits

Theorem 1.1 (Theorem of Rice (1953))

All non-trivial semantic questions about programs from a universal programming language are undecidable.

Example 1.2 (Detection of constants)

```
read(x);
if x > 0 then
    P; y := x;
else
    y := 1;
end;
write(y);
```

```
read(x);
if x > 0 then
    P; y := x;
else
    y := 1;
end;
write(1);
```

write(y) equivalently replaceable by write(1) iff program $P$ does never terminate

Thus: constant detection is undecidable
Introduction

Two Solutions

1. **Weaker models:**
   - employ *abstract models* of systems
     - finite automata, labeled transition systems, ...
   - perform *exact analyses*
     - model checking, theorem proving, ...

2. **Weaker analyses** (here):
   - employ *concrete models* of systems
     - source code
   - perform *approximate analyses*
     - dataflow analysis, abstract interpretation, type checking, ...
Introduction

Soundness vs. Completeness

• **Soundness:**
  – Predicted results must apply to every system execution
  – Examples:
    ■ constant detection: replacing expression by appropriate constant does not change program results
    ■ pointer analysis: analysis finds pointer variable $x \neq \text{null}$ $\implies$ no run-time exception when dereferencing $x$
  – Absolutely mandatory for trustworthiness of analysis results!

• **Completeness:**
  – Behavior of every system execution caught by analysis
  – Examples:
    ■ program always terminates $\implies$ analysis must be able to detect
    ■ value of variable in $[0, 255] \implies$ interval analysis finds out
  – Usually not guaranteed due to approximation
  – Degree of completeness determines precision of analysis

• **Correctness** := Soundness $\land$ Completeness
  (often for logical axiomatizations and such, usually not guaranteed for program analyses)
Introduction

Scalability and Practicability

- **Scalability:**
  - realistic programs can be handled with reasonable effort

- **Practicability:**
  - minimal specification effort and comprehensible results
The Imperative Model Language WHILE

Syntactic Categories

**WHILE**: simple imperative programming language without procedures or advanced data structures

**Syntactic categories**:

<table>
<thead>
<tr>
<th>Category</th>
<th>Domain</th>
<th>Meta variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>( \mathbb{Z} = {0, 1, -1, \ldots} )</td>
<td>( z )</td>
</tr>
<tr>
<td>Truth values</td>
<td>( \mathbb{B} = {\text{true, false}} )</td>
<td>( t )</td>
</tr>
<tr>
<td>Variables</td>
<td>( \text{Var} = {x, y, \ldots} )</td>
<td>( x )</td>
</tr>
<tr>
<td>Arithmetic expressions</td>
<td>( \text{AExp} ) (next slide)</td>
<td>( a )</td>
</tr>
<tr>
<td>Boolean expressions</td>
<td>( \text{BExp} ) (next slide)</td>
<td>( b )</td>
</tr>
<tr>
<td>Commands (statements)</td>
<td>( \text{Cmd} ) (next slide)</td>
<td>( c )</td>
</tr>
</tbody>
</table>
The Imperative Model Language WHILE

Syntax of WHILE Programs

Definition 1.3 (Syntax of WHILE)

The syntax of WHILE Programs is defined by the following context-free grammar:

\[ a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \ast a_2 \in AExp \]
\[ b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2 \in BExp \]
\[ c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } b \text{ do } c \text{ end} \in Cmd \]

Remarks: we assume that

- the syntax of numbers, truth values and variables is predefined (i.e., no “lexical analysis”)
- the syntax of ambiguous constructs is uniquely determined (by brackets, priorities, or indentation)
The Imperative Model Language WHILE

A WHILE Program and its Flow Diagram

Example 1.4

\[
\begin{align*}
x &:= 6; \\
y &:= 7; \\
z &:= 0; \\
\text{while } x > 0 \text{ do} \\
&\quad x := x - 1; \\
&\quad y := 7; \\
&\quad v := y; \\
&\quad \text{while } v > 0 \text{ do} \\
&\quad &\quad v := v - 1; \\
&\quad &\quad z := z + 1 \\
&\quad \text{end} \\
&\text{end}
\end{align*}
\]

Effect: \( z := x \times y = 42 \)
Overview of the Lecture

(Preliminary) Overview of Contents

1. Introduction to Program Analysis
2. Dataflow analysis (DFA)
   i. Available expressions problem
   ii. Live variables problem
   iii. The DFA framework
   iv. Solving DFA equations
   v. The meet-over-all-paths (MOP) solution
   vi. Case study: Java bytecode verifier
3. Abstract interpretation (AI)
   i. Working principle
   ii. Program semantics & correctness
   iii. Galois connections
   iv. Instantiations (sign analysis, interval analysis, ...)
   v. Case study: 16-bit multiplication
4. Interprocedural analysis
5. Pointer analysis
Additional Literature

• some papers (cf. web page)