Static Program Analysis
Lecture 1: Introduction to Program Analysis

Thomas Noll
Lehrstuhl für Informatik 2
(Software Modeling and Verification)

noll@cs.rwth-aachen.de

http://moves.rwth-aachen.de/teaching/ws-1415/spa/

Winter Semester 2014/15
1 Preliminaries

2 Introduction

3 The Imperative Model Language WHILE

4 Overview of the Lecture

5 Additional Literature
People

Lectures:
- Thomas Noll (noll@cs.rwth-aachen.de)
- Christina Jansen (christina.jansen@cs.rwth-aachen.de)

Exercise classes:
- Christian Dehnert (dehnert@cs.rwth-aachen.de)
- Benjamin Kaminski (benjamin.kaminski@cs.rwth-aachen.de)

Student assistant:
- Frederick Prinz
Target Audience

- MSc Informatik:
  - Theoretische Informatik
- MSc Software Systems Engineering:
  - Theoretical Foundations
What **you** can expect:

- Foundations of static analysis of computer software
- Implementation and tool support
- Applications in, e.g., program optimization and software validation
What you can expect:

- Foundations of static analysis of computer software
- Implementation and tool support
- Applications in, e.g., program optimization and software validation

What we expect: basic knowledge in

- Programming (essential concepts of imperative and object-oriented programming languages and elementary programming techniques)
- helpful: Theory of Programming (such as Semantics of Programming Languages or Software Verification)
Organization

Schedule:

- Lecture Mon 14:15–15:45 AH 1 (starting October 13)
- Lecture Thu 14:15–15:45 AH 2 (starting October 23)
- Exercise class Mon 10:15–11:45 AH 6 (starting October 27)
- see overview at http://moves.rwth-aachen.de/teaching/ws-1415/spa/
Schedule:
- Lecture Mon 14:15–15:45 AH 1 (starting October 13)
- Lecture Thu 14:15–15:45 AH 2 (starting October 23)
- Exercise class Mon 10:15–11:45 AH 6 (starting October 27)
- see overview at http://moves.rwth-aachen.de/teaching/ws-1415/spa/

1st assignment sheet next week, presented October 27

Work on assignments in groups of two
Schedule:
- Lecture Mon 14:15–15:45 AH 1 (starting October 13)
- Lecture Thu 14:15–15:45 AH 2 (starting October 23)
- Exercise class Mon 10:15–11:45 AH 6 (starting October 27)
- see overview at http://moves.rwth-aachen.de/teaching/ws-1415/spa/

1st assignment sheet next week, presented October 27
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
Schedule:
- Lecture Mon 14:15–15:45 AH 1 (starting October 13)
- Lecture Thu 14:15–15:45 AH 2 (starting October 23)
- Exercise class Mon 10:15–11:45 AH 6 (starting October 27)
- see overview at http://moves.rwth-aachen.de/teaching/ws-1415/spa/

1st assignment sheet next week, presented October 27

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
1 Preliminaries

2 Introduction

3 The Imperative Model Language WHILE

4 Overview of the Lecture

5 Additional Literature
Static (Program) Analysis

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

RWTH AACHEN
Dream of Static Program Analysis

Program Analyzer

Result

Property specification
Theorem 1.1 (Theorem of Rice (1953))

All non-trivial semantic questions about programs from a universal programming language are undecidable.
Theorem 1.1 (Theorem of Rice (1953))

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

Example 1.2 (Detection of constants)

\begin{verbatim}
read(x);
if x > 0 then
  P;
  y := x;
else
  y := 1;
end;
write(y);
\end{verbatim}

\begin{verbatim}
read(x);
if x > 0 then
  P;
  y := x;
else
  y := 1;
end;
write(1);
\end{verbatim}

\texttt{write(y)} can be equivalently replaced by \texttt{write(1)} iff program \texttt{P} does never terminate.
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);

write(y) can be equivalently replaced by write(1)
iff program P does never terminate

Thus: constant detection is undecidable
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, ...

Static Program Analysis

Winter Semester 2014/15 1.11
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, ...
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 $x \neq 0$ \implies no run-time exception when dereferencing $x$
- Absolutely mandatory for trustworthiness of analysis results!

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

Correctness $= \text{Soundness} \land \text{Completeness}$ (often for logical axiomatizations and such, usually not guaranteed for program analyses)
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 $x \neq 0$ $\implies$ 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 quality of analysis
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 \( x \neq 0 \) \( \implies \) 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 quality of analysis

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

1 Preliminaries

2 Introduction

3 The Imperative Model Language WHILE

4 Overview of the Lecture

5 Additional Literature
WHILE: simple imperative programming language without procedures or advanced data structures
**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>
Definition 1.3 (Syntax of WHILE)

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

\[ a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \ast a_2 \in AExp \]
\[ b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2 \in BExp \]
\[ c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \in Cmd \]
Definition 1.3 (Syntax of WHILE)

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

\[ a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \ast a_2 \in AExp \]

\[ b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2 \in BExp \]

\[ c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \in Cmd \]

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)
Example 1.4

```plaintext
x := 6;
y := 7;
z := 0;
while x > 0 do
    x := x - 1;
    v := y;
    while v > 0 do
        v := v - 1;
        z := z + 1
    STOP
```

Effect: $z := x \times y = 42$
Example 1.4

x := 6;
y := 7;
z := 0;
while x > 0 do
  x := x - 1;
v := y;
while v > 0 do
  v := v - 1;
z := z + 1
Example 1.4

\[ x \leftarrow 6; \]
\[ y \leftarrow 7; \]
\[ z \leftarrow 0; \]
\[ \textbf{while} \ x > 0 \ \textbf{do} \]
\[ \ \ x \leftarrow x - 1; \]
\[ \ v \leftarrow y; \]
\[ \textbf{while} \ v > 0 \ \textbf{do} \]
\[ \ \ v \leftarrow v - 1; \]
\[ \ z \leftarrow z + 1 \]

\[ \text{Effect: } z \leftarrow x \times y = 42 \]
Outline

1. Preliminaries
2. Introduction
3. The Imperative Model Language WHILE
4. Overview of the Lecture
5. Additional Literature
1. Introduction to Program Analysis

2. Dataflow analysis (DFA)
   1. Available expressions problem
   2. Live variables problem
   3. The DFA framework
   4. Solving DFA equations
   5. The meet-over-all-paths (MOP) solution
   6. Case study: Java bytecode verifier

3. Abstract interpretation (AI)
   1. Working principle
   2. Program semantics & correctness
   3. Galois connections
   4. Instantiations (sign analysis, interval analysis, ...)
   5. Case study: 16-bit multiplication

4. Interprocedural analysis

5. Pointer analysis
Outline

1 Preliminaries

2 Introduction

3 The Imperative Model Language WHILE

4 Overview of the Lecture

5 Additional Literature
[available in CS Library]

Michael I. Schwartzbach: *Lecture Notes on Static Analysis*  

[available in CS Library]