



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

Outline of Lecture 1

Preliminaries

Introduction

The Imperative Model Language WHILE

Overview of the Lecture

Additional Literature

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

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

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, ...)

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/>

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/>
- **1st assignment sheet** this Thursday, submitted & presented October 26
- Work on assignments in **groups of two**

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/>
- **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

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/>
- **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

Outline of Lecture 1

Preliminaries

Introduction

The Imperative Model Language WHILE

Overview of the Lecture

Additional Literature

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**.

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)

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, ...)

Introduction

Dream of Static Program Analysis

Program

```
socket.error: [Errno 111] Connection refused
print "ncfiles: Socket error (%s) for host %s (%s)" % (errno, strerror), host, (host, port))

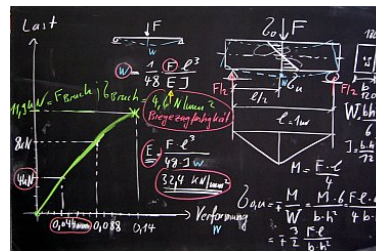
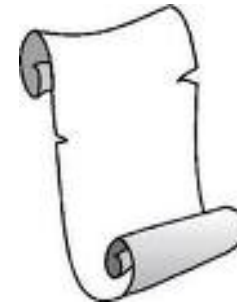
for h3 in page.findAll("h3"):
    value = (h3.contents[0])
    if value != "Afdeling":
        print >> txt, value
        import codecs
        f = codecs.open("alle.txt", "r", encoding="utf-8")
        text = f.read()
        f.close()
        # open the file again for writing
        f = codecs.open("alle.txt", "w", encoding="utf-8")
        f.write(value+"\n")
        # write the original contents
```



Analyzer



Result



Property specification

Fundamental Limits

Theorem 1.1 (Theorem of Rice (1953))

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

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

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**

Two Solutions

1. Weaker models:

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

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 $x \neq \text{null}$
 - \implies no run-time exception when dereferencing x
- Absolutely mandatory for **trustworthiness** of analysis results!

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

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 \wedge Completeness

(often for logical axiomatizations and such, usually not guaranteed for program analyses)

Scalability and Practicability

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

Scalability and Practicability

- **Scalability:**
 - realistic programs can be handled with reasonable effort
- **Practicability:**
 - minimal specification effort and comprehensible results

The Imperative Model Language WHILE

Outline of Lecture 1

Preliminaries

Introduction

The Imperative Model Language WHILE

Overview of the Lecture

Additional Literature

The Imperative Model Language WHILE

Syntactic Categories

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

The Imperative Model Language WHILE

Syntactic Categories

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

Syntactic categories:

Category	Domain	Meta variable
Numbers	$\mathbb{Z} = \{0, 1, -1, \dots\}$	z
Truth values	$\mathbb{B} = \{\text{true}, \text{false}\}$	t
Variables	$Var = \{x, y, \dots\}$	x
Arithmetic expressions	$AExp$ (next slide)	a
Boolean expressions	$BExp$ (next slide)	b
Commands (statements)	Cmd (next slide)	c

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:

$$a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$$
$$b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$$
$$c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } b \text{ do } c \text{ end} \in Cmd$$

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:

$$a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$$
$$b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$$
$$c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } b \text{ do } c \text{ end} \in Cmd$$

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

Example 1.4

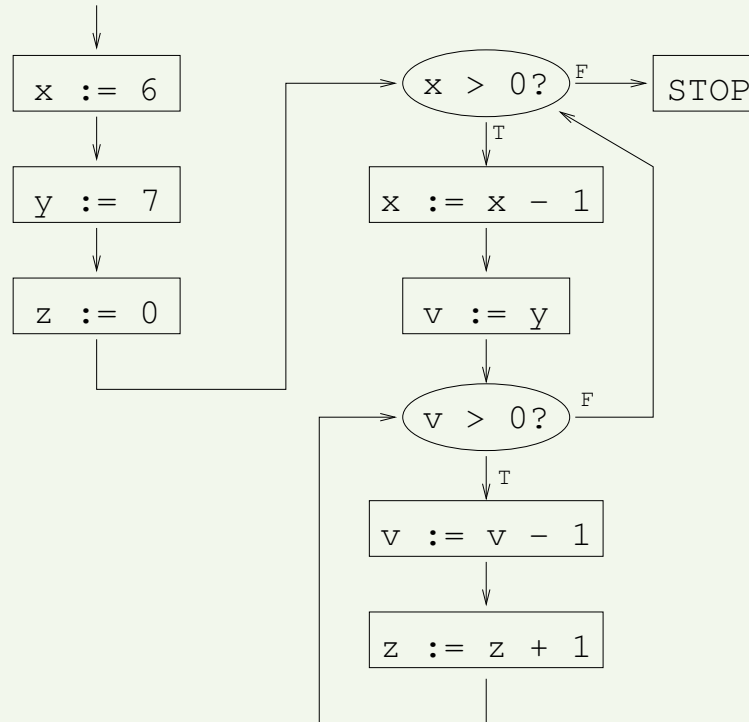
```
x := 6;
y := 7;
z := 0;
while x > 0 do
  x := x - 1;
  v := y;
  while v > 0 do
    v := v - 1;
    z := z + 1
  end
end
end
```

The Imperative Model Language WHILE

A WHILE Program and its Flow Diagram

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  
  end  
end  
end
```

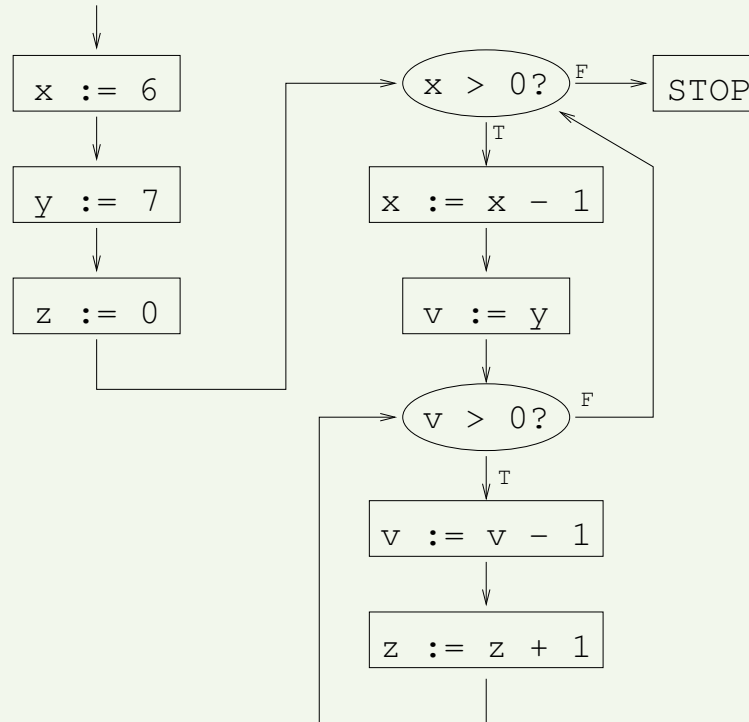


The Imperative Model Language WHILE

A WHILE Program and its Flow Diagram

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  
  end  
end  
end
```



Effect: $z := x * y = 42$

Overview of the Lecture

Outline of Lecture 1

Preliminaries

Introduction

The Imperative Model Language WHILE

Overview of the Lecture

Additional Literature

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

Outline of Lecture 1

Preliminaries

Introduction

The Imperative Model Language WHILE

Overview of the Lecture

Additional Literature

Additional Literature

Additional Literature

- Flemming Nielson, Hanne R. Nielson, Chris Hankin: **Principles of Program Analysis**, 2nd edition, Springer, 2005 [available in CS Library]
- Michael I. Schwartzbach: **Lecture Notes on Static Analysis**
[<http://www.itu.dk/people/brabrand/UFPE/Data-Flow-Analysis/static.pdf>]
- Helmut Seidl, Reinhard Wilhelm, Sebastian Hack: **Übersetzerbau 3: Analyse und Transformation**, Springer, 2010 [available in CS Library]
- some papers (cf. web page)