Static Program Analysis

Lecture 1: Introduction to Program Analysis

Summer Semester 2018

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-18/spa/

Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Outline of Lecture 1

People

- Lectures:
  - Thomas Noll (`noll@cs.rwth-aachen.de`)
- Exercise classes:
  - Christoph Matheja (`matheja@cs.rwth-aachen.de`)
- Student assistant:
  - Hannah Arndt

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, ...)

Organisation

- Schedule:
  - Lecture Mon 14:15–15:45 AH 6 (starting April 16)
  - Lecture Tue 14:15–15:45 AH 2 (starting April 17)
  - Exercise class Tue 12:15–13:45 AH 2 (starting April 24)
  - see overview at `https://moves.rwth-aachen.de/teaching/ss-18/spa/`

Organisation

- Schedule:
  - Lecture Mon 14:15–15:45 AH 6 (starting April 16)
  - Lecture Tue 14:15–15:45 AH 2 (starting April 17)
  - Exercise class Tue 12:15–13:45 AH 2 (starting April 24)
  - see overview at `https://moves.rwth-aachen.de/teaching/ss-18/spa/`
- 1st assignment sheet this week, submitted & presented April 24
- Work on assignments in groups of two

Organisation

- Schedule:
  - Lecture Mon 14:15–15:45 AH 6 (starting April 16)
  - Lecture Tue 14:15–15:45 AH 2 (starting April 17)
  - Exercise class Tue 12:15–13:45 AH 2 (starting April 24)
  - see overview at `https://moves.rwth-aachen.de/teaching/ss-18/spa/`
- 1st assignment sheet this week, submitted & presented April 24
- 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

Organisation

- Schedule:
  - Lecture Mon 14:15–15:45 AH 6 (starting April 16)
  - Lecture Tue 14:15–15:45 AH 2 (starting April 17)
  - Exercise class Tue 12:15–13:45 AH 2 (starting April 24)
  - see overview at `https://moves.rwth-aachen.de/teaching/ss-18/spa/`
- 1st assignment sheet this week, submitted & presented April 24
- 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

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)

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)

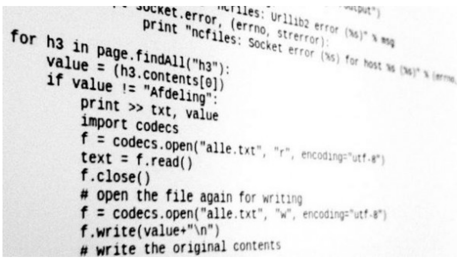
(Main) Applications:

Optimising 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

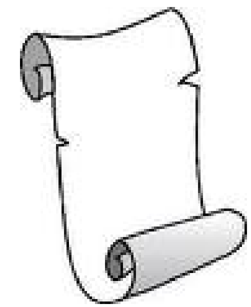


→

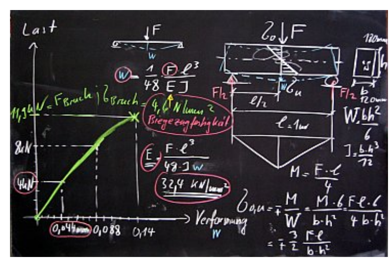
Analyzer

→

Result



↑



Property specification

Fundamental Limits

Theorem 1.1 (Theorem of Rice (1953))

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

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

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

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

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

Thus: constant detection is undecidable

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Two Solutions

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

Two Solutions

1. **Weaker models:**
   - employ abstract models of systems
     - finite automata, labelled 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, ...

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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 $x \neq \texttt{null}$
    $\Longrightarrow$ 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 $x \neq \texttt{null}$
    $\Longrightarrow$ 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 $\Longrightarrow$ analysis must be able to detect
  - value of variable in $[0, 255]$ $\Longrightarrow$ interval analysis finds out
- Usually not guaranteed due to (necessary) approximation
- Degree of completeness determines precision of analysis

Correctness, Scalability, and Practicability

Correctness := Soundness $\wedge$ Completeness

- Often for logical axiomatisations and such, usually not guaranteed for program analyses

Correctness, Scalability, and Practicability

Correctness := Soundness $\wedge$ Completeness

- Often for logical axiomatisations and such, usually not guaranteed for program analyses

Scalability

- Realistic programs can be handled with reasonable effort

Correctness, Scalability, and Practicability

Correctness $:=$ Soundness $\wedge$ Completeness

- Often for logical axiomatisations and such, usually not guaranteed for program analyses

Scalability

- Realistic programs can be handled with reasonable effort

Practicability

- Minimal specification effort and comprehensible results

Outline of Lecture 1

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

Syntax of WHILE Programs

Definition 1.3 (Syntax of WHILE)

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

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

Syntax of WHILE Programs

Definition 1.3 (Syntax of WHILE)

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

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

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)

A WHILE Program

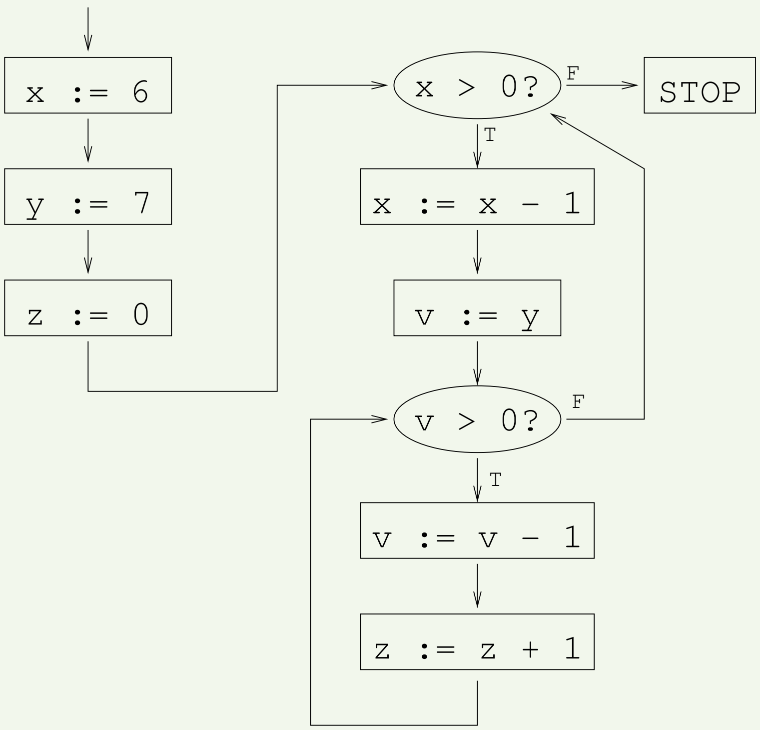
Example 1.4

```
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
```

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
```

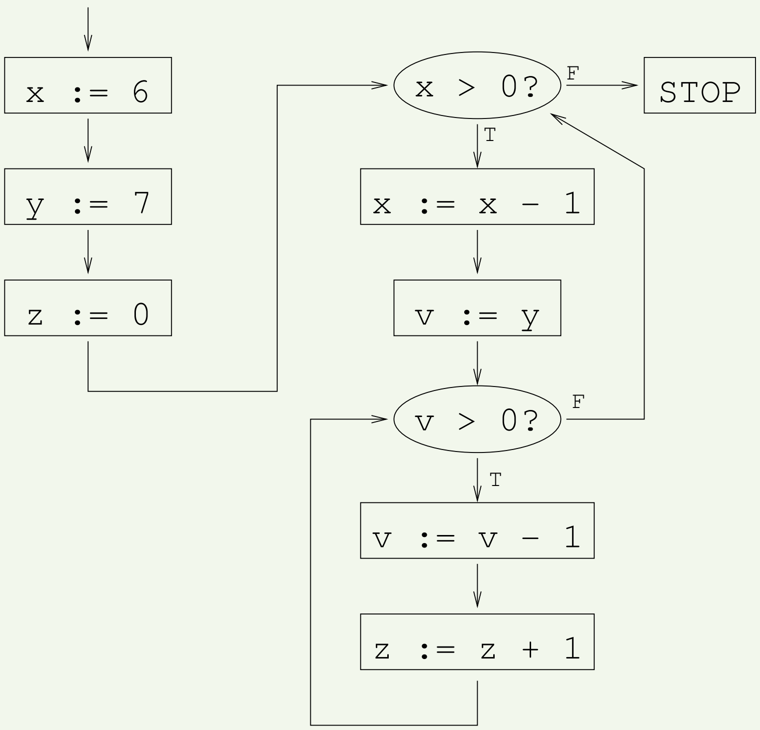


Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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
```



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

Outline of Lecture 1

Preliminaries

Introduction

The Imperative Model Language WHILE

Overview of the Lecture

Additional Literature

(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. Counterexample-Guided Abstraction Refinement (CEGAR)
   vi. Case study: 16-bit multiplication
4. Interprocedural analysis
5. Shape analysis

Outline of Lecture 1

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 and web links (cf. course web page)