Static Program Analysis

Lecture 1: Introduction to Program Analysis

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/spa11/

Summer Semester 2011

Outline

People

- Lectures: Thomas Noll
    - Lehrstuhl für Informatik 2, Room 4211
    - noll@cs.rwth-aachen.de
- Exercise classes:
    - Jonathan Heinen (heinen@cs.rwth-aachen.de)
    - Christina Jansen (christina.jansen@cs.rwth-aachen.de)
- Student assistant:
    - Stefan Breuer

Target Audience

- MSc Informatik:
    - Theoretische Informatik
- MSc Software Systems Engineering:
    - Theoretical CS
    - Specialization *Formal Methods, Programming Languages and Software Validation*
- Diplomstudiengang Informatik:
    - Theoretische Informatik
    - Vertiefungsfach *Formale Methoden, Programmiersprachen und Softwarevalidierung*
    - Combination with Katoen, Thomas, Vöcking, ...

Expectations

- 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 Wed 10:00–11:30 AH 6 (starting April 6)
  - Lecture Thu 15:00–16:30 AH 5 (bi-weekly; starting April 14)
  - Exercise class Wed 10:00–11:30 AH 2 (starting April 18)
  - see overview at `http://www-i2.informatik.rwth-aachen.de/i2/spa11/`
- 0th assignment sheet next week, presented April 18
- Work on assignments in groups of three
- Oral exam on appointment
  - for MSc candidates (6 credits)
  - for Diplom candidates (Übungsschein)
- 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

What Is It All About?

Static (Program) Analysis

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

Distinguishing features:

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

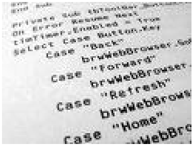
(Main) Applications:

Optimizing compilers: exploit program properties to improve runtime or memory efficiency of generated code (dead code elimination, constant propagation, ...)

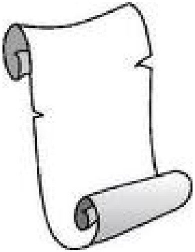
Software validation: verify program correctness (bytecode verification, shape analysis, ...)

Dream of Static Program Analysis

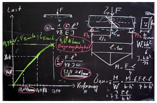
Program $\longrightarrow$ **Analyzer** $\longrightarrow$ **Result**



```
End Sub
Private Sub tbToolbar_
On Error Resume Next
tmrTimer.Enabled = True
Select Case Button.Key
    Case "Back"
        brwWebBrowser.Go
    Case "Forward"
        brwWebBrowser.
    Case "Refresh"
        brwWebBrows
    Case "Home"
        WebBro
```



$\uparrow$



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.

Example 1.2 (Detection of constants)

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

write(y) can be equivalently replaced by some constant 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, ...

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 $\implies$ 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 $\wedge$ Completeness
  (often for logical axiomatizations and such, usually not guaranteed for program analyses)

Outline

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	$Var = \{\mathtt{x}, \mathtt{y}, \ldots\}$	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:

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

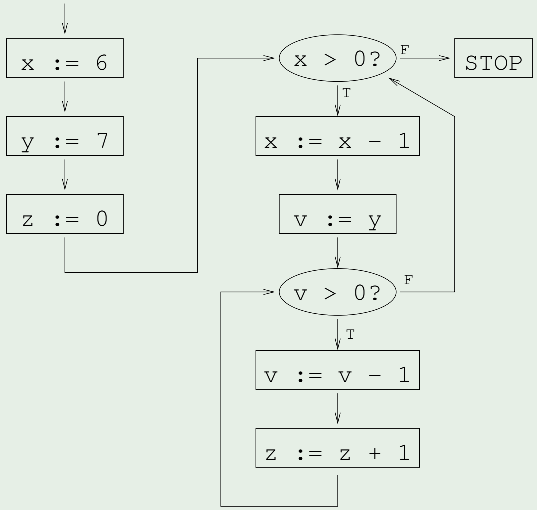
Remarks: we assume that

- 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)

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
```



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

Outline

(Preliminary) Overview of Contents

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: the Java bytecode verifier
3. Abstract interpretation (AI)
    1. Working principle
    2. Program semantics & correctness
    3. Galois connections
    4. Applications (sign analysis, interval analysis, ...)
4. Interprocedural analysis
5. Pointer analysis