Semantics and Verification of Software

Lecture 1: Introduction

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/svsw13/

Summer Semester 2013

Outline

Staff

- Lectures: Thomas Noll
  - Lehrstuhl für Informatik 2, Room 4211
  - E-mail `noll@cs.rwth-aachen.de`
- Exercise classes:
  - Kevin van der Pol (`kvdpol@cs.rwth-aachen.de`)
  - Stephen Wu (`Hao.Wu@cs.rwth-aachen.de`)
- Student assistants:
  - David Orlea

Target Audience

- Master[/Diplom] program Informatik
  - Theoretische Informatik
  - [Vertiefungsfach *Formale Methoden, Programmiersprachen und Softwarevalidierung* (Diplom)]
- Master program Software Systems Engineering
  - Theoretical CS

Target Audience

- Master[/Diplom] program Informatik
  - Theoretische Informatik
  - [Vertiefungsfach *Formale Methoden, Programmiersprachen und Softwarevalidierung* (Diplom)]
- Master program Software Systems Engineering
  - Theoretical CS
- In general:
  - interest in formal models for programming languages
  - application of mathematical reasoning methods
- Expected: basic knowledge in
  - essential concepts of imperative programming languages
  - formal languages and automata theory
  - mathematical logic

Organization

- Schedule:
    - Lecture Wed 10:00–11:30 AH 6 (starting Apr 10)
    - Lecture Thu 15:00–16:30 AH 5 (starting Apr 11)
    - Exercise class Mon 10:00–11:30 AH 2 (starting Apr 29)
- Irregular lecture dates – checkout web page!

Organization

- Schedule:
  - Lecture Wed 10:00–11:30 AH 6 (starting Apr 10)
  - Lecture Thu 15:00–16:30 AH 5 (starting Apr 11)
  - Exercise class Mon 10:00–11:30 AH 2 (starting Apr 29)
- Irregular lecture dates – checkout web page!
- 1st assignment sheet: next Monday (Apr 15) on web page
  - submission by Apr 22
  - presentation on Apr 29
- Work on assignments in groups of three

Organization

- Schedule:
  - Lecture Wed 10:00–11:30 AH 6 (starting Apr 10)
  - Lecture Thu 15:00–16:30 AH 5 (starting Apr 11)
  - Exercise class Mon 10:00–11:30 AH 2 (starting Apr 29)
- Irregular lecture dates – checkout web page!
- 1st assignment sheet: next Monday (Apr 15) on web page
  - submission by Apr 22
  - presentation on Apr 29
- Work on assignments in groups of three
- Examination (6 ECTS credits):
  - oral or written (depending on number of participants)
  - date to be fixed
- Admission requires at least 50% of the points in the exercises
- Solutions to exercises and exam in English or German

Outline

Aspects of Programming Languages

Syntax: "How does a program look like?"

- hierarchical composition of programs from structural components

$\Rightarrow$ *Compiler Construction*

Aspects of Programming Languages

Syntax: "How does a program look like?"

- hierarchical composition of programs from structural components

$\Rightarrow$ *Compiler Construction*

Semantics: "What does this program mean?"

- output/behavior/... in dependence of input/environment/...

$\Rightarrow$ This course

Aspects of Programming Languages

Syntax: "How does a program look like?"

- hierarchical composition of programs from structural components

$\Rightarrow$ *Compiler Construction*

Semantics: "What does this program mean?"

- output/behavior/... in dependence of input/environment/...

$\Rightarrow$ This course

Pragmatics:

- length and understandability of programs
- learnability of programming language
- appropriateness for specific applications, ...

$\Rightarrow$ *Software Engineering*

Aspects of Programming Languages

Syntax: "How does a program look like?"

- hierarchical composition of programs from structural components

$\Rightarrow$ *Compiler Construction*

Semantics: "What does this program mean?"

- output/behavior/... in dependence of input/environment/...

$\Rightarrow$ This course

Pragmatics:

- length and understandability of programs
- learnability of programming language
- appropriateness for specific applications, ...

$\Rightarrow$ *Software Engineering*

Historic development:

- Formal syntax since 1960s (LL/LR parsing); semantics defined by compiler/interpreter
- Formal semantics since 1970s (operational/denotational/axiomatic)

Why Semantics?

Idea: compiler = ultimate semantics!

- Compiler gives each individual program a semantics
  (= "behaviour" of generated machine code)

Why Semantics?

Idea: compiler = ultimate semantics!

- Compiler gives each individual program a semantics (= "behaviour" of generated machine code)

But:

- Compilers are highly complicated software systems (optimisations, interaction with runtime system, ...)
- Most languages have more than one compiler (with different outputs)
- Most compilers have bugs

$\Rightarrow$ Does not help with formal reasoning about programming language or individual programs

The Semantics of “Semantics”

Originally: study of meaning of symbols (linguistics)

Semantics of a program: meaning of a concrete program (I/O mapping, communication behavior, ...)

Semantics of a programming language: mapping of each (syntactically correct) program of a concrete programming language to its meaning

Semantics of software: various techniques for defining the semantics of diverse programming languages

Motivation for Rigorous Formal Treatment I

Example 1.1

1. How often will the following loop be traversed?

```
for i := 2 to 1 do ...
```

FORTRAN IV: once
PASCAL: never

Motivation for Rigorous Formal Treatment I

Example 1.1

1. How often will the following loop be traversed?

```
for i := 2 to 1 do ...
```

   FORTRAN IV: once
   PASCAL: never

2. What if `p = nil` in the following program?

```
while p <> nil and p^.key < val do ...
```

   Pascal: strict boolean operations ↯
   Modula: non-strict boolean operations ✓

Motivation for Rigorous Formal Treatment II

- Support for development of
  - new programming languages: missing details, ambiguities and inconsistencies can be recognized
  - compilers: automatic compiler generation from appropriately defined semantics
  - programs: exact understanding of semantics avoids uncertainties in the implementation of algorithms

Motivation for Rigorous Formal Treatment II

- Support for development of
  - new programming languages: missing details, ambiguities and inconsistencies can be recognized
  - compilers: automatic compiler generation from appropriately defined semantics
  - programs: exact understanding of semantics avoids uncertainties in the implementation of algorithms
- Support for correctness proofs of
  - programs: comparison of program semantics with desired behavior (e.g., termination properties, absence of deadlocks, ...)
  - compilers:

$$\begin{array}{rcl} \text{programming language} & \xrightarrow{\text{compiler}} & \text{machine code} \\ \text{semantics} \downarrow & & \downarrow \text{(simple) semantics} \\ \text{meaning} & \stackrel{?}{=} & \text{meaning} \end{array}$$

  - optimizing transformations:

$$\begin{array}{rcl} \text{code} & \xrightarrow{\text{optimization}} & \text{code} \\ \text{semantics} \downarrow & & \downarrow \text{semantics} \\ \text{meaning} & \stackrel{?}{=} & \text{meaning} \end{array}$$

(Complementary) Kinds of Formal Semantics

Operational semantics: describes computation of the program on some (very) abstract machine (G. Plotkin)

- example: $(\mathsf{seq})\ \dfrac{\langle c_1, \sigma\rangle \to \sigma' \quad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1 ; c_2, \sigma\rangle \to \sigma''}$
- application: implementation of programming languages (compilers, interpreters, ...)

(Complementary) Kinds of Formal Semantics

Operational semantics: describes computation of the program on some (very) abstract machine (G. Plotkin)

- example: $(\text{seq})\dfrac{\langle c_1, \sigma\rangle \to \sigma' \quad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1 ; c_2, \sigma\rangle \to \sigma''}$
- application: implementation of programming languages (compilers, interpreters, ...)

Denotational semantics: mathematical definition of input/output relation of the program by induction on its syntactic structure (D. Scott, C. Strachey)

- example: $\mathfrak{C}[\![.]\!] : Cmd \to (\Sigma \dashrightarrow \Sigma)$
  $\mathfrak{C}[\![c_1 ; c_2]\!] := \mathfrak{C}[\![c_2]\!] \circ \mathfrak{C}[\![c_1]\!]$
- application: program analysis

(Complementary) Kinds of Formal Semantics

Operational semantics: describes computation of the program on some (very) abstract machine (G. Plotkin)

- example: (seq) $\dfrac{\langle c_1, \sigma \rangle \to \sigma' \quad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1 ; c_2, \sigma \rangle \to \sigma''}$
- application: implementation of programming languages (compilers, interpreters, ...)

Denotational semantics: mathematical definition of input/output relation of the program by induction on its syntactic structure (D. Scott, C. Strachey)

- example: $\mathfrak{C}[\![.]\!] : Cmd \to (\Sigma \dashrightarrow \Sigma)$
  $\mathfrak{C}[\![c_1 ; c_2]\!] := \mathfrak{C}[\![c_2]\!] \circ \mathfrak{C}[\![c_1]\!]$
- application: program analysis

Axiomatic semantics: formalization of special properties of programs by logical formulae (assertions/proof rules; R. Floyd, T. Hoare)

- example: (seq) $\dfrac{\{A\}\, c_1 \,\{C\} \quad \{C\}\, c_2 \,\{B\}}{\{A\}\, c_1 ; c_2 \,\{B\}}$
- application: program verification

Overview of the Course

1. The imperative model language WHILE
2. Operational semantics of WHILE
3. Denotational semantics of WHILE
4. Equivalence of operational and denotational semantics
5. Axiomatic semantics of WHILE
6. Extensions: procedures and dynamic data structures
7. Applications: compiler correctness etc.

Literature

(also see the collection [“Handapparat”] at the CS Library)

- Formal semantics:
  - G. Winskel: *The Formal Semantics of Programming Languages*, The MIT Press, 1996
- Compiler correctness
  - H.R. Nielson, F. Nielson: *Semantics with Applications: A Formal Introduction*, Wiley, 1992

Outline

Syntactic Categories

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

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	$\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$	t
Variables	$\mathit{Var} = \{\mathtt{x}, \mathtt{y}, \ldots\}$	x
Arithmetic expressions	AExp (next slide)	a
Boolean expressions	BExp (next slide)	b
Commands (statements)	Cmd (next slide)	c

Syntax of WHILE Programs

Definition 1.2 (Syntax of WHILE)

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

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

Syntax of WHILE Programs

Definition 1.2 (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

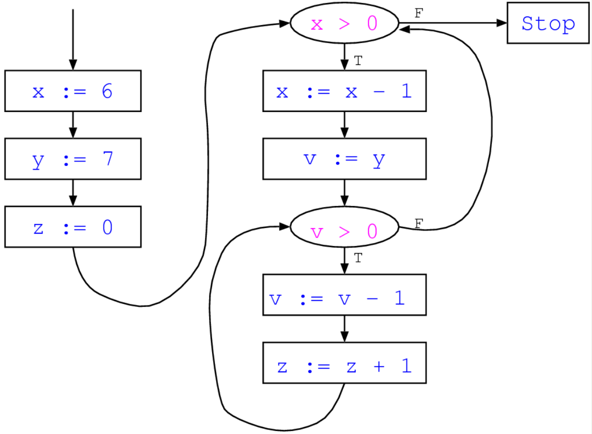
Example 1.3

```
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
```

A WHILE Program and its Flow Diagram

Example 1.3

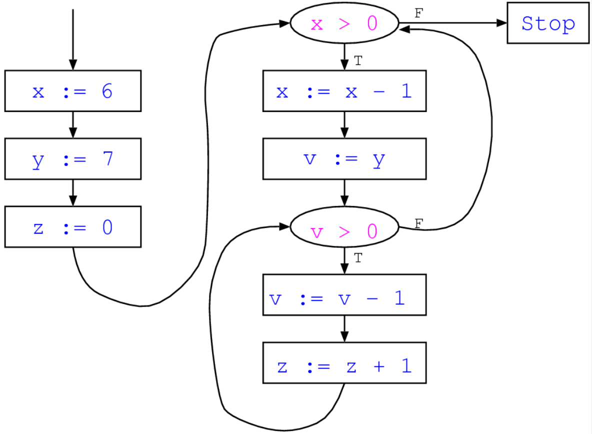
```
x := 6;
y := 7;
z := 0;
while x > 0 do
  x := x - 1;
  v := y;
  while v > 0 do
    v := v - 1;
    z := z + 1
```



A WHILE Program and its Flow Diagram

Example 1.3

```
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`