



Semantics and Verification of Software

Summer Semester 2015

Lecture 1: Introduction

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<http://moves.rwth-aachen.de/teaching/ss-15/sv-sw/>

Staff

- Lectures: **Thomas Noll**
 - Lehrstuhl für Informatik 2, Room 4211
 - E-mail noll@cs.rwth-aachen.de
- Exercise classes:
 - **Christoph Matheja** (matheja@cs.rwth-aachen.de)
 - **Federico Olmedo** (federico.olmedo@cs.rwth-aachen.de)
- Student assistants:
 - **Frederick Prinz**

Target Audience

- **MSc Informatik:**
 - Theoretische Informatik
- **MSc Software Systems Engineering:**
 - Theoretical Foundations of SSE
- 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**

Organisation

- Schedule:
 - **Lecture** Tue 14:15–15:45 AH 2 (starting 14 April)
 - **Lecture** Thu 11:15–12:45 AH 2 (starting 9 April)
 - **Exercise class** Wed 15:00–16:30 AH 6 (starting 22 April)
- Irregular lecture dates – checkout web page!
- Introductory exercise on 22 April
- 1st assignment sheet: next Wednesday (15 April) on web page
 - submission by 22 April
 - presentation on 29 April
- 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
- Written material in **English**, lecture and exercise classes “on demand”, rest up to you

Aspects of Programming Languages

Syntax: “How does a program look like?”

- hierarchical composition of programs from structural components
- ⇒ *Compiler Construction*

Semantics: “What does this program mean?”

- output/behaviour/... in dependence of input/environment/...
- ⇒ *This course*

Pragmatics: • *length* and *understandability* of programs

- *learnability* of programming language
 - *appropriateness* for specific applications, ...
- ⇒ *Software Engineering*

Historic development:

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

Introduction

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
 - code optimisations
 - memory management
 - interaction with runtime system
 - ...
 - Most languages have **more than one** compiler (with different outputs)
 - Most compilers have **bugs**
- ⇒ 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

- mapping input \rightarrow output values
- interaction behaviour (shared variables, communication, ...)
- ...

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

- operational
- denotational
- axiomatic
- ...

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?

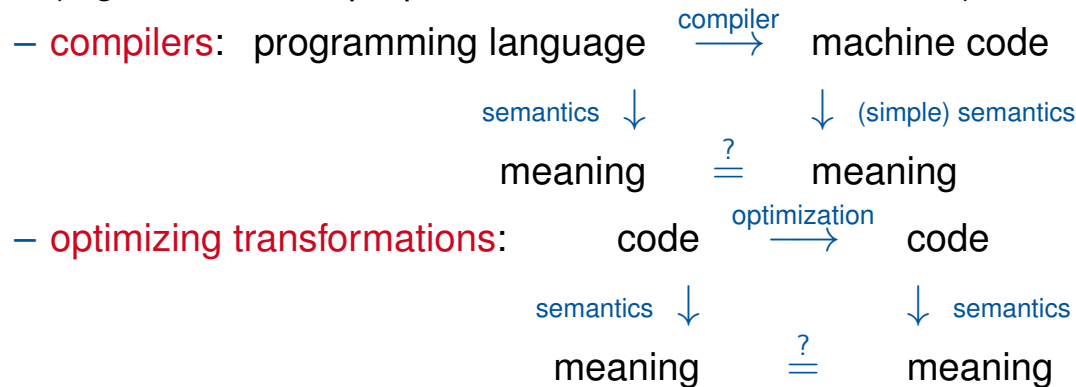
```
while p <> nil and p^.key < val do ...
```

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
- Support for **correctness proofs** of
 - **programs**: comparison of program semantics with desired behavior (e.g., termination properties, absence of deadlocks, ...)



Introduction

(Complementary) Kinds of Formal Semantics

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

- example:
$$\frac{\langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma''}{\langle c_1 ; c_2, \sigma \rangle \rightarrow \sigma''} \text{ (seq)}$$
- 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:
$$\begin{aligned} \mathcal{E}[\cdot] &: Cmd \rightarrow (\Sigma \dashrightarrow \Sigma) \\ \mathcal{E}[c_1 ; c_2] &:= \mathcal{E}[c_2] \circ \mathcal{E}[c_1] \end{aligned}$$
- application: program **analysis**

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

- example:
$$\frac{\{A\} c_1 \{C\} \quad \{C\} c_2 \{B\}}{\{A\} c_1 ; c_2 \{B\}} \text{ (seq)}$$
- 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.

Introduction

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: An Appetizer*, Springer Undergraduate Topics in Computer Science, 2007

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 1.2 (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

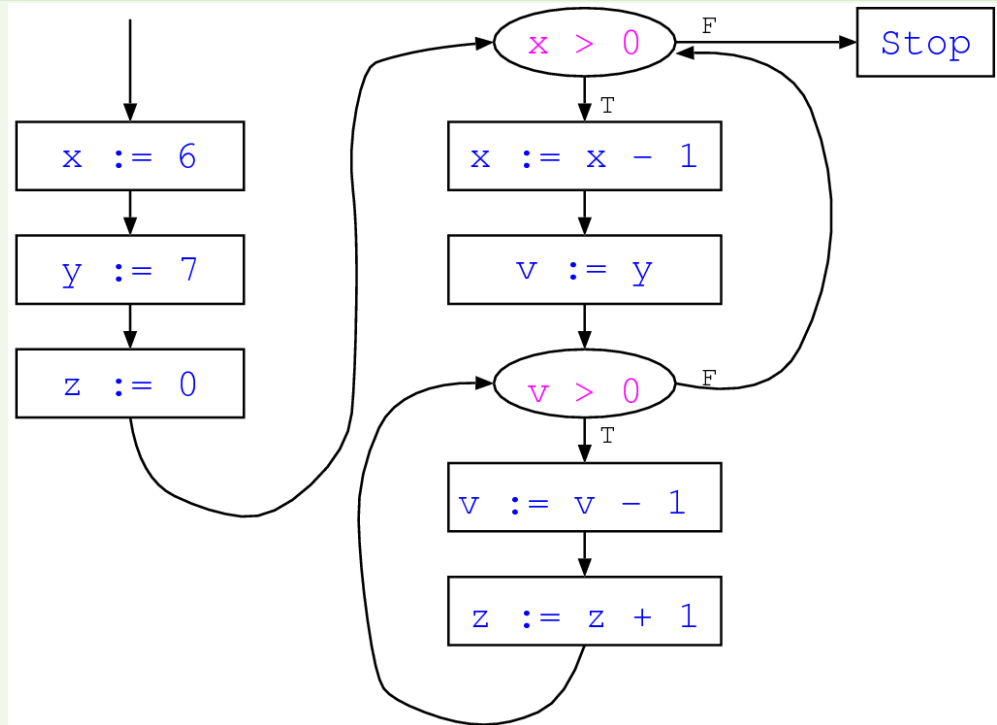
- the syntax of numbers, truth values and variables is predefined (i.e., no “lexical analysis”)
- the syntactic interpretation of ambiguous constructs (expressions) is uniquely determined (by brackets or priorities)

The Imperative Model Language WHILE

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  
  end  
end  
end
```



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