Programming Language Theory

Ralf Lämmel
Programming Language Theory

Introduction

Ralf Lämmel
Motivation
The Java grammar (or the C# or Cobol grammars for that matter) has hundreds of productions. How can we possibly abstract from this complexity and understand the essence of OO programming (likewise for other paradigms)?
What are the different programming paradigms anyhow? What multi-paradigmatic combinations are there?
What is a reasonable style to define the meaning of language concepts?

- Any way you like?
- Visitor style of OO programming?
- Functional programming?
- ...

© Ralf Lämmel, 2009-2013 unless noted otherwise
What does it really mean to be statically or dynamically typed? How do semantics and type system interrelate?
What heavy lifting can formal treatment of programming languages provide?
Programming language theory =
Formal semantics +
Programming paradigms
Formal semantics [Wikipedia]:

In theoretical computer science, formal semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages and models of computation. The formal semantics of a language is given by a mathematical model that describes the possible computations described by the language.
Formal semantics - why?

- Theory of programming languages is the mathematical study of the meaning of programs.

- The goal is to find ways to describe program behaviors that are both precise and abstract.

  - **precise** so that we can use mathematical tools to formalize and check interesting properties, and

  - **abstract** so that properties of interest can be discussed clearly, without getting bogged down in low-level details.

This slide is derived from [Odersky]; see resources for the course.
Formal semantics - why?

- **To develop intuitions for informal reasoning about programs.**

- **To understand language features and their interactions** and thereby also develop principles for better language design (PL is the “materials science” of computer science...).

- **To prove general facts about all the programs in a given programming language** (e.g., safety or isolation properties).

- **To prove specific properties of particular programs** (i.e., program verification), which is important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but it is still quite difficult and expensive.
Application areas of formal semantics

- Language understanding
- Language prototyping
- Compiler construction
- Program verification
- Program analysis
- Program optimization
- Program translation
- ...

This slide is derived from [Nielson]; see resources for the course.
Classic approaches to formal semantics

- **Denotational semantics** and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.

- **Axiomatic semantics** is based on program logics such as Hoare logic and dependent type theories focus on logical rules for reasoning about programs.

- **Operational semantics** describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.

- **Process calculi** focus on the communication and synchronization behaviors of complex concurrent systems.

- **Type systems** describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

- ...
Programming paradigm [Wikipedia]:

A programming paradigm is a fundamental style of computer programming. (Compare with a methodology, which is a style of solving specific software engineering problems). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints, etc.) and the steps that compose a computation (assignation, evaluation, continuations, data flows, etc.).
(Multi-) paradigms

- Logic programming
- Functional programming
- Imperative programming
- Object-oriented programming
- Constraint-logic programming
- Concurrent programming
- Parallel programming
- ...
Programming paradigms
(incl. “multi”-paradigms)

- Logic programming
- Functional programming
- Imperative programming
- Object-oriented programming
- Constraint-logic programming
- Concurrent programming
- Parallel programming
- ...
(Multi-) paradigms in the course

- **Logic programming**
  - Prerequisite: foundations & primitive, if any, programming skills.
  - Become proficient as a Prolog programmer.
  - Use Prolog as a 1st class sandbox for formal semantics.

- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**
(Multi-) paradigms in the course

- Logic programming
- **Functional programming**
  - Use Haskell as a “2nd class” sandbox for formal semantics.
  - Better understand the pros (and cons) of static typing.
- Imperative programming
- Object-oriented programming
- Constraint-logic programming
- Concurrent programming
- Parallel programming
(Multi-) paradigms in the course

- Logic programming
- Functional programming
- **Imperative programming**
  - Prerequisite: related OO programming skills.
  - Serves as a basic studying subject for formal semantics.
- Object-oriented programming
- Constraint-logic programming
- Concurrent programming
- Parallel programming
(Multi-) paradigms in the course

- Logic programming
- Functional programming
- Imperative programming
- **Object-oriented programming**
  - Prerequisite: modest OO programming skills.
  - Serves as a challenging studying subject for formal semantics.
- Constraint-logic programming
- Concurrent programming
- Parallel programming
(Multi-) paradigms in the course

- Logic programming
- Functional programming
- Imperative programming
- Object-oriented programming
- **Constraint-logic programming**
  - Constraints are a powerful programming concept.
  - CLP is a matured multi-paradigm language with constraints.
  - We don’t have much time for it.
- Concurrent programming
- Parallel programming
(Multi-) paradigms in the course

- Logic programming
- Functional programming
- Imperative programming
- Object-oriented programming
- Constraint-logic programming
- **Concurrent programming**
  - Concurrency important for modern, complex systems.
  - Understand designated formal semantics: process calculi.
- Parallel programming
(Multi-) paradigms in the course

- Logic programming
- Functional programming
- Imperative programming
- Object-oriented programming
- Constraint-logic programming
- Concurrent programming

- **Parallel programming**
  - Parallelism important for current hardware trends.
  - We may mention parallelism in the context of FP, perhaps.
Formal semantics: a short introduction
Formal semantics in context

- Theory of programming languages
  - Formal syntax
    - (E)BNF as a description formalism
    - Parsing as an execution method of (E)BNF
  - Formal semantics
    - Assign meaning to syntax
      - Interpret syntax in a stepwise manner
      - Interpret syntax in a compositional manner
The formal semantics approach

<table>
<thead>
<tr>
<th>concrete syntax</th>
<th>LL(1)/LALR(1) grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>concrete syntax trees</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>syntactic categories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>abstract syntax trees</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>semantics</th>
<th>semantic categories</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>semantic definitions</td>
</tr>
</tbody>
</table>

This slide is derived from [Nielson]; see resources for the course.
Syntactic categories of the **While** language

- **numerals**
  \[ n \in \text{Num} \]

- **variables**
  \[ x \in \text{Var} \]

- **arithmetic expressions**
  \[ a \in \text{Aexp} \]
  \[ a ::= n \mid x \mid a_1 + a_2 \]
  \[ \quad \mid a_1 \ast a_2 \mid a_1 - a_2 \]

- **booleans expressions**
  \[ b \in \text{Bexp} \]
  \[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \]
  \[ \quad \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \]

- **statements**
  \[ S \in \text{Stm} \]
  \[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \]
  \[ \quad \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \]
  \[ \quad \mid \text{while } b \text{ do } S \]
Operational semantics

\[ y := 1; \]
\[ \text{while } \neg (x = 1) \text{ do } (y := x \times y; x := x - 1) \]

First we assign 1 to \( y \), then we test whether \( x \) is 1 or not. If it is then we stop and otherwise we update \( y \) to be the product of \( x \) and the previous value of \( y \) and then we decrement \( x \) by one. Now we test whether the new value of \( x \) is 1 or not \( \cdots \)
Operational semantics works with configurations of the form

\[
\langle \text{control}, \text{data} \rangle.
\]

Roughly:

- \textit{control} – “where are we”, \textit{data} – the values of program variables.

\textit{control} may be absent (final configuration).

Structural Operational Semantics

Sequences of configurations, \( \text{conf}_1 \Rightarrow \text{conf}_2 \Rightarrow \cdots \).

(Small step semantics.)

Natural Semantics (big step semantics)

\[\langle \text{control}, \text{data} \rangle \rightarrow \text{data}' \quad \text{in one step.}\]
Denotational semantics

\[ y := 1; \]
\[ \text{while } \neg(x = 1) \text{ do } (y := x \times y; x := x - 1) \]

The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of \( x \) will be 1 and the value of \( y \) will be equal to the factorial of the value of \( x \) in the initial state.

Two kinds of denotational semantics:

- Direct Style Semantics
- Continuation Style Semantics
Axiomatic semantics

```plaintext
y := 1;
while ¬(x = 1) do (y := x * y; x := x - 1)
```

If $x = n$ holds before the program is executed then $y = n!$ will hold when the execution terminates (if it terminates)

Two kinds of axiomatic semantics:

- Partial Correctness
- Total Correctness
Course metadata

http://softlang.wikidot.com/course:plt
Course - size & hours & points

Lecture
2 SWS

Lab
2 SWS

6 LP
This course

• **Get involved!**

• See exam rules on the website.

• See other metadata there as well.

http://softlang.wikidot.com/course:plt
Resources for this lecture


See the website for more resource description for the course.

http://softlang.wikidot.com/course:plt
Leveling expectations:
What not to expect from this course ...

- The most formal part of formal semantics.
- A classic logic programming course.
- An introduction to functional programming.

http://softlang.wikidot.com/course:plt
• **Summary**: Tough content. Doable exam.

• **Prepping**:
  - Start reading Nielson\(^2\) or Pierce!
  - Get up to speed with Prolog today!

• **Lab**: Prolog skills

• **Outlook**: Formal semantics