Semantyka i weryfikacja programów

Andrzej Tarlecki

Instytut Informatyki
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

`http://www.mimuw.edu.pl/~tarlecki` pok. 4750
`tarlecki@mimuw.edu.pl` tel: (22 55) 44475, 44214

Strona tego wykładu: `http://www.mimuw.edu.pl/~tarlecki/teaching/semwer/`

Program Semantics & Verification

Andrzej Tarlecki

Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

`http://www.mimuw.edu.pl/~tarlecki` office: 4750
`tarlecki@mimuw.edu.pl` phone: (48)(22)(55) 44475, 44214

This course: `http://www.mimuw.edu.pl/~tarlecki/teaching/semwer/`

Overall

- The aim of the course is to present the importance as well as basic problems and techniques of formal description of programs.
- Various methods of defining program semantics are discussed, and their mathematical foundations as well as techniques are presented.
- The basic notions of program correctness are introduced together with methods and formalisms for their derivation.
- The ideas of systematic development of correct programs are introduced.

Prerequisites

Current version:

- Wstęp do programowania (1000-211bWPI, 1000-211bWPF)
- Podstawy matematyki (1000-211bPM)

Old version:

- Wstęp do programowania (1000-211WPI, 1000-211WPF)
- Wstęp do teorii mnogości (1000-211WTM)
- Logika (1000-212LOG)

Literature

Rather random choice for now:

- P. Dembiński, J. Małuszyński. *Matematyczne metody definiowania języków programowania*. WNT, 1981.
- M. Gordon. *Denotacyjny opis języków programowania*. WNT, 1983.
- H. Riis Nielson, F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1999.
- D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
- E. Dijkstra. *Umiejętność programowania*. WNT, 1978.

Programs

```
D207 0C78 F0CE 00078 010D0        r := 0; q := 1;
D203 0048 F0D6 00048 01CD8        while q <= n do
8000 F0EA F0B3 010EC 00ED7          begin r := r + 1;
9C00 000C F0DA 0000C ...                  q := q + 2 * r + 1 end
```

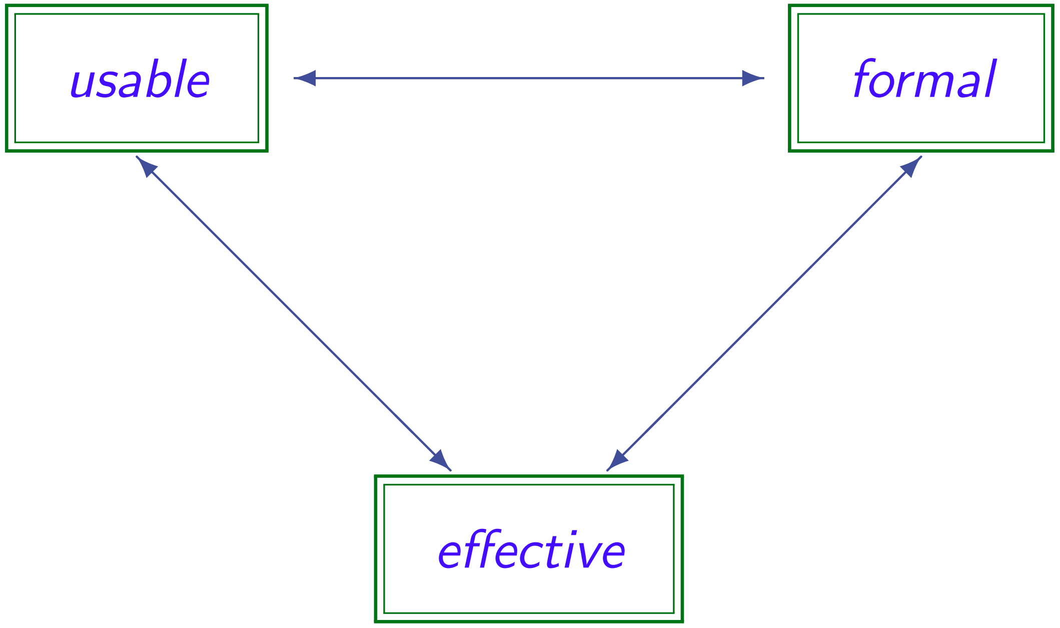
- a precise description of an *algorithm*, understandable for a human reader
- a precise prescription of *computations* to be performed by a computer

Programs should be:

- clear; efficient; robust; reliable; user friendly; well documented; . . .
- but first of all, *CORRECT*
- don't forget though: also, *executable*. . .

Tensions

A triangle of tension for programming languages:



Grand View

What we need for a good programming language:

- Syntax
- Semantics
- Logic
- Pragmatics/methodology
- Implementation
- Programming environment

Syntax

To determine exactly the well-formed phrases of the language.

– *concrete syntax* (LL(1), LR(1), . . .)

– *abstract syntax* (CF grammar, BNF notation, etc)

– *type checking* (context conditions, static analysis)

It is standard by now to present it formally!

One consequence is that excellent tools to support parsing are available.

Semantics

To determine the meaning of the programs and all the phrases of the language.

Informal description is often not good enough

– operational semantics (small-step, big-step, machine-oriented): dealing with the notion of *computation*, thus indicating *how* the results are obtained

– denotational semantics (direct-style, continuation-style): dealing with the overall *meaning* of the language constructs, thus indicating the results without going into the details of how they are obtained

– axiomatic semantics: centred around the *properties* of the language constructs, perhaps ignoring some aspects of their meanings and the overall results

Pragmatics

To indicate how to use the language well, to build *good* programs.

– user-oriented presentation of programming constructs

– hints on good/bad style of their use

Logic

To express and prove program properties.

- Partial correctness properties, based on first-order logic
- Hoare's logic to prove them
- Termination properties (total correctness)

Also:

– temporal logics

– other modal logics

– algebraic specifications

– abstract model specifications

program verification vs. *correct program development*

Methodology

- specifications
- stepwise refinement
- designing the modular structure of the program
- coding individual modules

Implementation

Compiler/interpreter, with:

- parsing
- static analysis and optimisations
- code generation

Programming environment

So that we can actually do this:

- dedicated text/program editor
- compiler/interpreter
- debugger
- libraries of standard modules

BUT ALSO:

- support for writing specifications
- verification tool
- . . .

Why **formal** semantics?

So that we can sleep at night. . .

– precise understanding of all language *constructs* and the underlying *concepts*

– independence of any particular implementation

– easy prototype implementations

– necessary basis for trustworthy reasoning

Example

Recall:

```
r := 0; q := 1;
while q <= n do
  begin r := r + 1;
        q := q + 2 * r + 1
  end
```

Or better:

$$
\begin{aligned}
& rt := 0;\, sqr := 1; \\
& \mathbf{while}\ sqr \leq n\ \mathbf{do}\ (rt := rt + 1; \\
& \qquad\qquad\qquad\qquad sqr := sqr + 2 * rt + 1)
\end{aligned}
$$

Well, this computes the integer square root of n, doesn't it:

```
$\{n \geq 0\}$
$rt := 0; sqr := 1;$
$\{n \geq 0 \wedge rt = 0 \wedge sqr = 1\}$
$\textbf{while } \{sqr = (rt+1)^2 \wedge rt^2 \leq n\}\ sqr \leq n \textbf{ do}$
        $(rt := rt + 1;$
        $\{sqr = rt^2 \wedge sqr \leq n\}$
        $sqr := sqr + 2 * rt + 1)$
$\{rt^2 \leq n < (rt+1)^2\}$
```

But how do we justify the implicit use of assertions and proof rules?

Sample proof rule

For instance:

$$\{sqr = rt^2 \wedge sqr \leq n\}\ sqr := sqr + 2 * rt + 1\ \{sqr = (rt+1)^2 \wedge rt^2 \leq n\}$$

follows by:

$$\boxed{\{\varphi[E/x]\}\ x := E\ \{\varphi\}}$$

BUT: although correct *in principle*, this rule fails in quite a few ways for PASCAL (abnormal termination, looping, references and sharing, side effects, assignments to array components, etc)

$$\boxed{\textit{Be formal and precise!}}$$

Justification

- definition of program semantics
- definition of satisfaction for correctness statements
- proof rules for correctness statements
- proof of soundness of all the rules
- analysis of completeness of the system of rules

Course outline

- Introduction
- Operational semantics
- Denotational semantics for simple and somewhat more advanced constructs
- Foundations of denotational semantics
- Partial correctness: Hoare's logic
- Total correctness: proving termination
- Systematic program derivation
- Semantics: an algebraic view (with bits and pieces of universal algebra)
- Program specification and development

Syntax

There are standard ways to define a syntax for programming languages. The course to learn about this:

Języki, automaty i obliczenia

Basic concepts:

- *formal languages*
- (generative) *grammars*: regular (somewhat too weak), *context-free* (just right), context-dependent (too powerful), . . .

BTW: there are grammar-based mechanisms to define the semantics of programming languages: attribute grammars, perhaps also two-level grammars, see (or rather, go to)

Metody implementacji języków programowania

Concrete syntax

Concrete syntax of a programming language is typically given by a (context-free) grammar detailing all the "commas and semicolons" that are necessary to write a string of characters that is a well-formed program. Typically, there are also additional context dependent conditions to eliminate some of the strings permitted by the grammar (like "thou shalt not use an undeclared variable").

Presenting a formal language by an unambiguous context-free grammar gives a *structure* to the strings of the language: it shows how a well-formed string is build of its immediate components using some linguistic *construct* of the language.

Abstract syntax

Abstract syntax presents the structure of the program phrases in terms of the linguistic constructs of the language, by indicating the *immediate components* of the phrase and the *construct* used to build it.

Think of abstract syntax as presenting each phrase of a language as a tree: the node is labelled by the top construct used, with the subtrees giving the immediate components.

Parsing is the way to map concrete syntax to abstract syntax, by building the abstract syntax tree for each phrase of the language as defined by the concrete syntax.

All these concepts (and more) are explained at other courses.

At this course

We will not belabour the distinction between concrete and abstract syntax.

- concrete-like way of presenting the syntax will be used
- the phrases will be used as if they were given by an abstract syntax
- if doubts arise, parenthesis and indentation will be used to disambiguate the interpretation of a phrase as an abstract-syntax tree

This is inappropriate for true programming languages

but quite adequate to deal with our examples