Concrete Semantics
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2021-10-6
Chapter 1

Introduction
1 Background

2 This Course
1 Background

2 This Course
Why Semantics?

Without semantics, we do not really know what our programs mean. We merely have a good intuition and a warm feeling. Like the state of mathematics in the 19th century — before set theory and logic entered the scene.
Intuition is important!

• You need a good intuition to get your work done efficiently.
• To understand the average accounting program, intuition suffices.
• To write a bug-free accounting program may require more than intuition!
• I assume you have the necessary intuition.
• This course is about “beyond intuition”.
Intuition is not sufficient!

Writing correct language processors (e.g. compilers, refactoring tools, . . .) requires

• a deep understanding of language semantics,
• the ability to reason (= perform proofs) about the language and your processor.

Example:
What does the correctness of a type checker even mean? How is it proved?
Why Semantics??

We have a compiler — that is the ultimate semantics!!

- A compiler gives each individual program a semantics.
- It does not help with reasoning about the PL or individual programs.
- Because compilers are far too complicated.
- They provide the worst possible semantics.
- Moreover: compilers may differ!
The sad facts of life

- Most languages have one or more compilers.
- Most compilers have bugs.
- Few languages have a (separate, abstract) semantics.
- If they do, it will be informal (English).
Bugs

- Google “compiler bug”
- Google “hostile applet”
  Early versions of Java had various security holes. Some of them had to do with an incorrect bytecode verifier.

GI Dissertationspreis 2003:
Gerwin Klein: *Verified Java Bytecode Verification*
Standard ML (SML)

First real language with a mathematical semantics:
Milner, Tofte, Harper:
The Definition of Standard ML. 1990.

Robin Milner (1934–2010)
Turing Award 1991.

Main achievements:  LCF (theorem proving)
                    SML (functional programming)
                    CCS, pi (concurrency)
The sad fact of life

SML semantics hardly used:

• too difficult to read to answer simple questions quickly
• too much detail to allow reliable informal proof
• not processable beyond \LaTeX, not even executable
More sad facts of life

- Real programming languages *are* complex.
- Even if designed by academics, not industry.
- Complex designs are error-prone.
- Informal mathematical proofs of complex designs are also error-prone.
The solution

Machine-checked language semantics and proofs

- Semantics at least type-correct
- Maybe executable
- Proofs machine-checked

The tool:

Proof Assistant (PA)
or
Interactive Theorem Prover (ITP)
Proof Assistants

• You give the structure of the proof
• The PA checks the correctness of each step
• Can prove hard and huge theorems

Government health warnings:

Time consuming
Potentially addictive
Undermines your naive trust in informal proofs
Formal = machine-checked
Verification = formal correctness proof

Formal = mathematical
Two landmark verifications

C compiler
Competitive with gcc -01
Xavier Leroy
INRIA Paris
using Coq

Operating system
microkernel (L4)
Gerwin Klein (& Co)
NICTA Sydney
using Isabelle
A happy fact of life

Programming language researchers are increasingly using PAs
Why verification pays off

Short term: \textit{The software works!}

Long term:

\begin{itemize}
\item Tracking effects of changes by rerunning proofs
\item Incremental changes of the software typically require only incremental changes of the proofs
\end{itemize}

Long term much more important than short term: \textit{Software Never Dies}
1 Background

2 This Course
What this course is *not* about

- Hot or trendy PLs
- Comparison of PLs or PL paradigms
- Compilers (although they will be one application)
What this course *is* about

- Techniques for the description and analysis of
  - PLs
  - PL tools
  - Programs
- Description techniques: *operational semantics*
- Proof techniques: *inductions*

*Both informally and formally (PA!)*
Our PA: Isabelle/HOL

- Started 1986 by Paulson (U of Cambridge)
- Later development mainly by Nipkow & Co (TUM) and Wenzel
- The logic HOL is ordinary mathematics

Learning to use Isabelle/HOL is an integral part of the course

All exercises require the use of Isabelle/HOL
Why I am so passionate about the PA part

- It is the future
- It is the only way to deal with complex languages **reliably**
- I want students to learn how to write correct proofs
- I have seen too many proofs that look more like LSD trips than coherent mathematical arguments
Overview of course

- Introduction to Isabelle/HOL
- IMP (assignment and while loops) and its semantics
- A compiler for IMP
- Hoare logic for IMP
- Type systems for IMP
- Program analysis for IMP
The semantics part of the course is mostly traditional
The use of a PA is leading edge
A growing number of universities offer related course
What you learn in this course goes far beyond PLs

It has applications in compilers, security, software engineering etc.

It is a new approach to informatics