Semantics of Programming Languages (1)

Christian Urban

Tuesdays 15:00 - 16:30, Zuse
Wednesdays 16:00 - 17:30, Zuse

http://www4.in.tum.de/~urbanc/Teaching/semantics08.html
Why Semantics?

We usually know what we intend with a program. Problem: How do we match this intention up what we write down?

For this we need precise definitions of what each language construct means (to exclude misunderstandings).
Why Semantics?

We usually know what we intend with a program. Problem: How do we match this intention up what we write down?

For this we need precise definitions of what each language construct means (to exclude misunderstandings).

Studying the semantics of programming languages is about solving this problem.
Why Semantics?

We usually know what we intend with a program. Problem: How do we match this intention up with what we write down?

For this we need precise definitions of what each language construct means (to exclude misunderstandings).

Studying the semantics of programming languages is about solving this problem.

In addition, we can look at design choices made in PLs; we can use semantics as a starting point for proving properties about programs; improve compilers...
Most programming languages are described (if at all) in natural language (that is English).

e.g. Java, C, C++, Ruby, Perl, Python, XML
Why All the Hard Work?

Most programming languages are described (if at all) in natural language (that is English).

This is one of the most stupid things you can do, because

- no way to exclude misunderstandings
- all too easy to mess things up
- no way to prove things about PLs and programs
Why All the Hard Work?

- Most programming languages are described (if at all) in natural language (that is English).

- This is one of the most stupid things you can do, because
  - no way to exclude misunderstandings
  - all too easy to mess things up
  - no way to prove things about PLs and programs

- We need a formal semantics.
It is Really Hard!

Formal semantics are hard to get right (PLs make promises):

- SML is a shining example of a PL that has a formal semantics, but it needed several attempts to get it right.

Robin Milner
Turing Award, 1991
It is Really Hard!

Formal semantics are hard to get right (PLs make promises):

- SML is a shining example of a PL that has a formal semantics, but it needed several attempts to get it right.

- Early versions of Java contained a "feature" in the classloader, which allowed you to break all its security promises.*

  * see “Hostile Applets” webpage
Complexity Kicks In

- PLs might be easy to learn (in contrast to natural languages)...

- ...but to get them right (ensure that all promises are indeed true) needs attention to a lot of details. We need proofs.
PLs might be easy to learn (in contrast to natural languages)...

...but to get them right (ensure that all promises are indeed true) needs attention to a lot of details. We need proofs.

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.
A: The definitions are almost always wrong.

“Anonymous” cited in B. Pierce’s book on Types and Programming Languages
This Course

You can take a course in semantics in probably all the top-100 universities in the world.

(According to the Shanghai ranking TUM is ranked 56th in 2007; 14th in Europe.)
This Course

You can take a course in semantics in probably all the top-100 universities in the world.
(According to the Shanghai ranking TUM is ranked 56th in 2007; 14th in Europe.)

However, we will do the proving part using a theorem prover (Isabelle/HOL).

This is cutting edge: I only know of courses at

- UPenn, Harvard, Princeton, probably some places in France, very few others.
You can take a course in semantics in probably all the top-100 universities in the world. (According to the Shanghai ranking TUM is ranked 56th in 2007; 14th in Europe.)

However, we will do the proving part using a theorem prover (Isabelle/HOL).

Warning: This is an experiment! It might crash and burn.

We owe it to you to be on the cutting edge.

This is cutting edge: I only know of courses at

- UPenn, Harvard, Princeton, probably some places in France, very few others.
What this Course Is Not About

- not about programming
- not about compilers (but we will take a look at compiling)
- not about the latest trendy/hot/fashionable PL
- not about comparing different PLs and not about finding out that “my” language is the best

Fifty years of programming language research, and we end up with C++?

(R. A. O’Keefe)
What this Course Is About

- learn about techniques for design, description and analysis of PLs
- learn about how to make and how to prove rigorous claims about PLs
- understand mathematical techniques in semantics (various inductions)
- learn about the lambda-calculus (just the simplest language for doing interesting exercises)
- learn how to use a theorem prover (Isabelle/HOL)
Administrative Stuff

Information - check regularly (slides etc)
http://www4.in.tum.de/~urbanc/Teaching/semantics08.html

Exam - at the end; written; accounts for 70% of the grade

Homework - very important
only accounts for 30%, but is essential for your understanding; it will be impossible to do well in the exam without taking the exercises seriously.

If helpful, I can start a Google forum.

(Exercises - TBD)
What is a Theorem Prover?

(We are here interested only in the ones that are interactive.)

- They expect that “hard steps” in a proof are done by humans (interactively); “easy steps” are sometimes found automatically.
- They are quite stupid (this makes you think harder about what you like to prove).
- They give you instant feedback.
- One can nowadays do quite impressive things with them (compilers, 4-colour thm.).
- They let you sleep better at night. ;o)
What is a Theorem Prover?
(We are here interested only in the ones that are interactive.)

They expect that “hard steps” in a proof are done by humans (interactively); “easy steps” are found automatically.

They are quite stupid (this makes you think harder about what you like to prove).

They give you instant feedback.

One can nowadays do quite impressive things (compilers, 4-colour thm.).

They let you sleep better at night. ;o)
What is a Theorem Prover?

(We are here interested only in the ones that are interactive.)

- They expect that “hard steps” in a proof are done by humans (interactively); “easy steps” are sometimes found automatically.

- They are quite stupid (this makes you think harder about what you like to prove).

- They give you instant feedback.

- One can nowadays do quite impressive things with them (compilers, 4-colour thm.).

- They let you sleep better at night. ;o)
Why Using Isabelle/HOL?

- I know about it ;o)
- It is developed mainly in Munich. (started in Cambridge 20 years ago)
- Isabelle/HOL is based on a small logical core, that is HOL, guaranteeing logical correctness. For me the HOL has the feel of “usual mathematics”.
- We will use the Nominal Datatype Package (my research) to deal with binders.

This package is only available in Isabelle/HOL.
Why Using Isabelle/HOL?

- I know about it ;o)
- It is developed mainly in Munich.
- Usually theorem provers are taught by introducing first what logic is about and what simple inferences are (very dull ;o).
- I will skip this and show you directly the real “thing” — just like a programming language is first introduced with a “hello-world” program.
- This is an experiment (I did tell you that already, didn’t I?).
Why Am I so Passionate About the Theorem Proving Part?

An often-heard criticism: Although you obtain more confidence in the correctness of your results, theorem provers make you work slower.
Why Am I so Passionate About the Theorem Proving Part?

An often-heard criticism: Although you obtain more confidence in the correctness of your results, theorem provers make you work slower.

My answer to this is: You get additional benefits too, namely you can experiment with your proofs and definitions.
My Experience

LF is a framework for representing and reasoning about logics and programming languages (mainly used at CMU).

Correctness in LF depends on type-checking, which in turn depends on type-equivalence checking.

Bob Harper  Frank Pfenning
(CMU)  (CMU)

They describe in a 41-pages paper an algorithm that decides type-equivalence.
We Had a Look at their Work

\textbf{def} \ \overset{\cong}{=} \ \textbf{Proof} \ \overset{\rightarrow}{\longrightarrow} \ \textbf{Alg.}

41pp in ACM Trans. on Comp. Logic (2005)
We Had a Look at their Work

\[ \text{def} \quad \text{Proof} \quad \text{Alg.} \]

41pp in ACM Trans. on Comp. Logic (2005)

\[ \text{def} = \text{Proof} \quad \text{Alg.} \]

1. solution (within 2h)
We Had a Look at their Work

41pp in ACM Trans. on Comp. Logic (2005)

1. solution (within 2h)

2. solution (suggested by Harper)
We Had a Look at their Work

41pp in ACM Trans. on Comp. Logic (2005)

1. solution (within 2h)

2. solution (suggested by Harper)

3. solution
We Had a Look at their Work

\[ \text{def} \quad \text{Proof} \quad \text{Alg.} \]

41pp in ACM Trans. on Comp. Logic (2005)

1. solution (within 2h)

2. solution (suggested by Harper)

Note: I also found errors in my PhD-thesis.
We Had a Look at their Work

One can do such things only with a theorem prover. The point is that you will be able to do the same, i.e. experiment with definitions and proofs. (This is fun ;o)
Another often-heard criticism: Theorem provers are not helpful for "real-world" problems.
Why Am I so Passionate About the Theorem Proving Part?

Another often-heard criticism: Theorem provers are not helpful for “real-world” problems.

My answer to this is: Just one impressive example...
Bugs in the compiler can lead to incorrect machine code generated from a correct program. (annoying)

They can invalidate any guarantees obtained by formal methods on the program level. (evil)
Bugs in the compiler can lead to incorrect machine code generated from a correct program. (annoying)

They can invalidate any guarantees obtained by formal methods on the program level. (evil)

Admittedly, the number of bugs in compilers is small compared to other programs, but...
But: Trojan Horses Can Be Easily Hidden in Compilers

Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code. (very readable “Reflections on Trusting Trust”)

Therefore looking at the source code will not protect you from such “Thompson-hacks”.

Ken Thompson
Turing Award, 1983
But: Trojan Horses Can Be Easily Hidden in Compilers

- Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.
  (very readable “Reflections on Trusting Trust”)

- Therefore looking at the source code will not protect you from such “Thompson-hacks”.

- You need a checkable certificate

Ken Thompson
Turing Award, 1983
he programmed an entire compiler for Cminor in a theorem prover

and proved:

Theorem:

For all programs $S$, if the compiler generates machine code $C$ from $S$ (without reporting an error), and if $S$ satisfies property $P$, then also $C$ satisfies $P$.

$P$ is observational behaviour / proof can be independently checked.
A Landmark Result

What is really exciting about his work is that the code generated for Cminor is only 15% slower than gcc -O1.

(his webpage: http://pauillac.inria.fr/~xleroy)
What is really exciting about his work is that the code generated for Cminor is only 15% slower than gcc -O1.

(his webpage: http://pauillac.inria.fr/~xleroy)

Also interesting for us:

The trust for the compiler is reduced to the problem of trusting the semantics of the source language (e.g. Cminor) and of the target language (e.g. PPC).
Why Am I so Passionate About the Theorem Proving Part?

“Real programmers don’t need abstract concepts to get their jobs done.”
Why Am I so Passionate About the Theorem Proving Part?

“Real programmers don’t need abstract concepts to get their jobs done.”

An answer to this is:

My prediction is that theorem provers will become powerful programming environments. For example when the user proves invariants about his/her code, then the compiler can translate this code more aggressively.
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics

What is the meaning of $3 + 4$?
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics

What is the meaning of

$3 + 4$

mathematical universe
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics

What is the meaning of

$3 + 4 = 7$

mathematical universe
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics

What is the meaning of

\[ 3 + 4 \]

mathematical universe
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics

What is the meaning of \( 3 + 4 \)?

\[
\begin{align*}
0 + n & \Downarrow n \\
S(n) + m & \Downarrow S(r)
\end{align*}
\]

(we only need to know what \( S \) is)
Styles of Semantics

- denotational semantics
- operational semantics
- axiomatic semantics

What is the meaning of $3 + 4$?

more complicated
Variables

What is the meaning of

\[ 3 + 4 + x \]
Variables

What is the meaning of

\[ 3 + 4 + x \] ?

for variables we need an environment

\[ [x \mapsto 1, y \mapsto 42, \ldots] \]
Let say we have

```
datatype aexp =
    V string
| C nat
| Diff aexp aexp ("--")
```

The expression $2 \rightarrow x$ is

$\text{(C 2)} \rightarrow \text{(V x)}$
A Language of Expressions

Let say we have

```
datatype aexp = V string
               | C nat
               | Diff aexp aexp ("--")
```

The expression $\mathbf{2} -- x$ is

$$(C \ 2) -- (V \ x)$$

$$(C \ 2) -- (C \ 3)$$ is also a legal expression. What could it mean?
Environments

An environment is a list of (string,nat)-pairs

types env = "(string × nat) list"

We can look up a variable in an env

fun lookup :: "string ⇒ env ⇒ nat option"
where
  "lookup x [] = None"
| "lookup x ((y,n)#tail) =
    (if x=y then (Some n) else (lookup x tail))"
Next Week

- We will write a little “compiler” for this expression language and machine.
- Prove that the compiler is correct (produces the same result as eval).
- Look at ways (monads) how to improve our definitions.
- (Later) Extend everything to include functions. This will be the interesting part.
Now to the difficult part ;o)

- Isabelle needs PolyML and (X)emacs
- easy to install under Linux and MacOSX
- difficult under Windows (not impossible, but needs cygwin or a “live image”)

- download and installation notes from:
  http://www.cl.cam.ac.uk/research/hvg/Isabelle/installation.html

- need this in the lectures
Now to the difficult part ;o)

- Isabelle needs PolyML and (X)emacs
- easy to install under Linux and MacOSX
- difficult under Windows (not impossible, but needs cygwin or a “live image”)

- download and installation notes from http://www.cl.cam.ac.uk/research/hvg/Isabelle/installation.html

- need this in the lectures
- your homework for this week is to get Isabelle/HOL running
Installation of Isabelle/HOL

Now to the difficult part ;o)

- Isabelle needs PolyML and (X)emacs
- easy to install under Linux and MacOSX
- difficult under Windows (not impossible, but needs cygwin or a "live image")

download and installation notes from

http://www.cl.cam.ac.uk/research/hvg/Isabelle/installation.html

need this in the lectures

your homework for this week is to get Isabelle/HOL running

My office is MI 00.09.053. Send me an email first.