Teaching Semantics with a Proof Assistant: No more LSD Trip Proofs

Tobias Nipkow

Institut für Informatik, Technische Universität München

Abstract. We describe a course on the semantics of a simple imperative programming language and on applications to compilers, type systems, static analyses and Hoare logic. The course is entirely based on the proof assistant Isabelle and includes a compact introduction to Isabelle. The overall aim is to teach the students how to write correct and readable proofs.

1 Introduction

A perennial challenge for both students and teachers of theoretical informatics courses are proofs and how to teach them. Scott Aaronson [1] characterizes the situation very well:

> I still remember having to grade hundreds of exams where the students started out by assuming what had to be proved, or filled page after page with gibberish in the hope that, somewhere in the mess, they might accidentally have said something correct.
>
> . . . innumerable examples of “parrot proofs” — NP- completeness reductions done in the wrong direction, arguments that look more like LSD trips than coherent chains of logic . . .

One could call it the London underground phenomenon:

Students Proofs

I do not want to play the blame game but want to suggest a way to bridge this gap with the help of a proof assistant. The underlying assumptions are that

- the above mentioned “LSD trip proofs” are the result of insufficient practice and that
- proof assistants lead to abundant practice because they are addictive like video games:

Let me explain the analogy between proof assistants and video games. The main advantage of a proof assistant is that it gives the students immediate feedback. Incorrect proofs are rejected right away.[1] Such rejections are an insult and challenge for motivated students. They will persist in their struggle to convince the machine, chasing that elusive `No subgoals`, Isabelle's equivalent of `You have reached the next level of World of Proofcraft`. Of course many students need the additional motivation that the homework they struggle with actually counts towards their final grade.

This is in contrast to the usual system of homework that is graded by a teaching assistant and returned a week later, long after the student struggled with it, and at a time when the course has moved on. This delay significantly reduces the impact that any feedback scribbled on the homework may have. Of course, a proof assistant does not replace a teaching assistant, who can explain *why* a proof is wrong and what to do about it. This is why lab sessions in the presence of teaching assistants are still essential.

The rest of the paper describes a new programming language semantics course based on the proof assistant Isabelle/HOL [14], where "semantics" really means "semantics and applications", for example compilers and program analyses. All of the material of the course (Isabelle theories, slides, lecture notes) are freely available at `http://www.in.tum.de/~nipkow/semantics`.

2 Course History and Format

Fifteen years ago I formalized parts of Winskel's textbook [23], which I was teaching from, in Isabelle/HOL. The longer term vision was a "Mechanized Semantics Textbook" [11, 12] as I called it. Although I used the growing collection of semantics theories in Isabelle in my courses, I did not teach Isabelle in my semantics courses and certainly did not require the students to write Isabelle proofs. I felt that the proof language available at the time was not suitable for expressing proofs at an abstract enough level for use in a semantics (as opposed to a proof assistant) course. Predictably, the students' ability to write proofs did not improve as a result. It was Christian Urban who first taught a semantics

[1] And a fair number of (morally) correct ones, too, because they lack the details the proof assistant unfortunately tends to require.

course at TUM based on Isabelle/HOL. He focused on λ-calculus and employed Nominal Isabelle/HOL [18] to deal with variable binding. He in turn was inspired by the Software Foundations course by Benjamin Pierce [17] who teaches Coq and selected areas of both imperative languages and λ-calculus. At this point Isabelle had acquired a readable proof language, which overcame my earlier reluctance to teach Isabelle in a semantics course. I designed a new semantics course and finally made the "Mechanized Semantics Textbook" (in the form of Isabelle theories) the basis of the new course. It is this new course that I report on.

2.1 Format

It is a graduate level course in the theory section of the curriculum. There are a number of alternative courses, for example Logic, Automata Theory, and Algorithms. The Semantics course typically attracts 15–20 students, most of them master students. There are 3 full lecture hours per week, and 1.5 hours of lab sessions, over 15 weeks. In the lab sessions, led by teaching assistants, the students are asked to solve exercises and the solutions are discussed at the end. In addition, there is a homework sheet every one or two weeks. The course is worth 8 ECTS points, with 30 ECTS points being the average workload per semester.

The whole course is based on the proof assistant Isabelle. It is used in the lectures, the lab sessions and in the homework. The homework is the heart and soul of the course. Hence 40% of the final grade is based on the homework. The exam, which contributes 60% to the final grade, is independent of Isabelle and focuses on semantics and informal proofs.

2.2 Prerequisites

Because this is a graduate level course (although advanced undergraduates take it, too), we expect that the students have some background in logical notation, proofs, and functional programming.

Logic We expect some basic familiarity with logical notation from introductory mathematics courses, typically a discrete math course. We assume that the students are able to read and understand simple formulas of predicate logic involving functions, sets and relations. They should have been exposed to mathematical proofs, including induction, but we do not expect that they can write such proofs themselves (reliably).

Functional Programming We assume that the students have had some exposure to functional programming, to the extent that they know about ML-style datatypes, recursive functions and pattern matching.[2]

[2] At TUM, a functional programming course is mandatory for informatics undergraduates.

3 Aims and Principles

This section explains the general aims and principles underlying the course; to a large extent they are generic and independent of the semantics content.

3.1 No More LSD Trip Proofs!

Next to semantics, this is the central aim of the course. Programming language semantics and its applications deal with complex objects, for example compilers. Analyzing such tools requires precise proofs. Graduate level informatics students who specialize in this area must understand the underlying proof principles and must be able to construct such proofs, both informally on paper and with the help of a machine.

We believe we have largely reached this aim. The students obtained on average 88% of the homework points, an unprecedented percentage that is due to the video game effect of a proof assistant described in the introduction together with the 40% incentive. In fact, any system where students can immediately tell how much of the homework they have solved successfully without having to hand it in will create a strong incentive to maximize the number of points obtained.

3.2 Teach Semantics, Not Proof Assistants

More precisely:

> Teach a Semantics course with the help of a proof assistant,
> not a Proof Assistant course with semantics examples.

The Semantics dog should wag the proof assistant tail, not the other way around. We believe we have reached that goal. Only approximately one quarter of the semester is dedicated to the proof assistant, the remaining three quarters are dedicated to semantics. Neither did we have to make any compromises on the semantics front. The material is no simpler than what we had covered in the past.

3.3 Teach Proofs, Not Proof Scripts

Most theorem provers provide a scripting language for writing proofs. Such proofs are sequences of commands to the prover that, in their entirety, are hard or impossible to read for the human, unless he executes them in the proof assistant. In Isabelle they look like this, where . . . elides some basic proof methods:

```
apply(. . . )
apply(. . . )
⋮
```

Such proofs are for machines, not for humans. They lack the information what is being proved at each point, and they lack structure. They do not convey ideas. They are like assembly language programs. Luckily, Isabelle has a higher-level proof language *Isar* [21] that is inspired by the proof language of Mizar [8] (see [22] for a comparison). Mizar has no low-level scripting language. Hence the "Teach proofs, not proof scripts" principle is really due to Mizar.

As an example we show an Isar proof of Cantor's theorem, where f is a function from a type to its powerset and ... are again suitable proof methods:

```
lemma ¬surj f
proof
  assume surj f
  hence ∃a. {x | x ∉ f x} = f a by ...
  hence False by ...
qed
```

This is the proof language used for most of the course. It is close to the informal language of mathematics and allows a smooth transition in the presentation style during the course: from Isar proofs on the machine to more traditional proofs on the blackboard (see Section 3.5).

3.4 Teach Proofs, Not Logic

Of course this is not a new idea, mathematicians have been doing this successfully for a long time. To be provocative: proof systems like natural deduction belong in logic courses, where the fine structure of logic is studied. But for students who already have some exposure to logical notation and proof (see Section 2.2), single step proofs in some proof system are a straightjacket. Application-oriented courses—remember, we are trying to teach Semantics, not logic— should reason *modulo logic*: if the student believes that A together with B implies C, he should be able to just write

from A **and** B **have** C **by** *hammer*

where *hammer* is some suitable proof method of the underlying proof assistant. Isar allows exactly that, and Isabelle offers a number of automatic hammers for this purpose, in particular the connection to powerful external automatic provers [4]. The motto is: Do not let logic dominate your thinking, let automatic provers take care of logic.

If *hammer* fails, the user has to refine the proof. This is exactly the Mizar approach. Of course there is a problem: how to figure out what intermediate step might help the proof assistant to see reason, or how to figure out that the claimed deduction is not valid? For the second alternative, Isabelle offers tools for counterexample search [4]. For the first alternative (and also the second!), we have to admit that proof scripts are an excellent way to home in on gaps in a proof. Hence we actually teach apply-scripts, but only in small doses. We also teach a bit of natural deduction, but in disguise: not as inference rules but as Isar text patterns. See Section 4.

3.5 Do Not Let the Proof Assistant Dominate Your Presentation

In the beginning of the course, when introducing the proof assistant, it is essential to demonstrate the interaction with the proof assistant in class for long periods. But even then, we intersperse these demos with slides that introduce or summarize *concepts* and go beyond what the interaction with the system will tell you. Moreover, displaying an Isabelle file with a video projector is never as pleasing to the eye as a separate presentation of the same material, say some function definitions, as LaTeXed slides. Isabelle's LaTeX generation facility and its "antiquotations" allow us to transfer material from Isabelle files to slides automatically without having to type it in a second time.

During the second part of the course, the Semantics part, we gradually move to conventional presentations based on slides and the blackboard, although we never completely abandon Isabelle. We believe that slides (with animations) and especially the blackboard are better suited to explain many concepts and proofs than an Isabelle demo is. When moving to the blackboard for developing proofs, we initially stick closely to Isar to phrase these proofs. As the students become more comfortable with Isar, we begin taking more and more liberties on the blackboard, moving towards informal proofs. The aim is to strengthen the students' ability to bridge the gap between formal and informal proofs.

3.6 Executability Matters

Most students' intuition is greatly enhanced by executable models. As it turns out, most of our formalizations are naturally executable. This is obvious for recursive functions but less so for inductively defined predicates. In fact, students at first do not think of inductive predicates as executable, unless they have a Prolog background. It is an important insight that, for example, inductively defined operational semantics is executable because of the dataflow from the initial to the final state. It is exactly with this application in mind that we made inductive predicates executable in Isabelle [3, 2], subject to a mode analysis. This unique feature of Isabelle permits us to execute most of the models in our course, no matter whether they are recursive or inductive or mixtures thereof.

4 Teaching Isabelle

In this and the next section we describe the technical content of the course: first the introduction to Isabelle and proofs, and then the actual semantics and applications part.

Following our principle that the proof assistant should only be a means to an end, namely teaching semantics, we introduce only as much of Isabelle as is necessary for the semantics material. This enables us to cover the material in about 4 weeks, just over a quarter of the semester. The details of this approach can be found elsewhere [13].

We follow the Isabelle tutorial [14] in making functional programming the entry road to theorem proving. HOL includes a functional language, just like other proof assistants do. We assume that the students have had some exposure to functional programming (see Section 2.2) and introduce HOL as a combination of programming and logic, starting with booleans, natural numbers and lists. Students who lack a functional programming background usually manage to pick up the principles from those examples. To start with, our only formulas are equations. After week 1, students can write examples like this:

```
datatype tree = Node tree nat tree | Tip

fun mirror :: tree ⇒ tree where
mirror (Node l n r) = Node (mirror r) n (mirror l) |
mirror Tip = Tip

lemma mirror (mirror t) = t
apply(induct t)
apply auto
done
```

Contrary to our motto "Teach proofs, not proof scripts" we introduce proof scripts after all. One reason is their succinctness: the syntax is minimal, which is important at this early stage where students are easily confused by all the new syntax. At the same time the students are taught what the corresponding informal proofs look like.

Week 2 offers a first taste of semantics. Week 1 has been an uphill struggle for the students. It is important show them that they can already model a number of interesting notions on a simple level. We introduce arithmetic and boolean expressions, their evaluation, constant folding optimization, and a compiler from arithmetic expressions to a stack machine. Of course we also prove that optimizer and compiler preserve the semantics.

Week 3 introduces logic beyond equality. We assume that the students are able to read and understand simple formulas of predicate logic (recall Section 2.2) and we merely explain how to write them in HOL. They are also introduced to Isabelle's array of automatic proof tools, in particular *Sledgehammer* [4], Isabelle's link to the automatic first-order provers E, SPASS, Vampire and Z3. Sledgehammer soon becomes the students' best friend in their battle with proofs. Since not all proofs are automatic, we also explain a limited amount of single step reasoning with apply-scripts, as motivated in Section 3.4. Inductive definitions are introduced as the last important modeling tool.

Week 4 is dedicated to the structured proof language Isar (see Section 3.3). In addition to Isar itself we also teach a number of useful proof patterns that correspond to natural deduction rules, for example

```
show ¬ P
proof
  assume P
  ⋮
  show False  ...
qed
```

Although Isabelle's automation subsumes single natural deduction steps, such patterns can improve readability of proofs if used selectively.

5 Semantics

The course concentrates on a single imperative language IMP as you can find it in traditional textbooks by Winskel [23] and Nielson and Nielson [9, 10]. In fact, we cover material similar to that by the Nielsons. There are two choices here. Concentrating on a single language permits us to cover many aspects and applications of semantics such as compilers, type systems, Hoare logic, static analyses and abstract interpretation. Concentrating on an imperative language builds on standard background knowledge of the students and emphasizes the relevance of semantics to mainstream computer science, thus facilitating the motivation of the students. We give a sketchy overview of the material. It was developed jointly with Gerwin Klein and more details can be found in the Isabelle distribution and in print [6].

5.1 IMP

IMP is the *de facto* standard imperative language in semantics courses. It contains arithmetic expressions (type *aexp*), boolean expressions (type *bexp*), and commands (type *com*). Commands are defined as the following datatype:

```
datatype com = SKIP
             | vname ::= aexp
             | com; com
             | IF bexp THEN com ELSE com
             | WHILE bexp DO com
```

Variable names (type *vname*) are strings. The state of an IMP program is a function from *vname* to *int*. Arithmetic and boolean expressions are evaluated by recursively defined functions. The only arithmetic operator in *aexp* is + and the only comparison operator in *bexp* is <. Commands are given both a big and a small step semantics and their equivalence is proved:

$$(c,s) \Rightarrow t \ \longleftrightarrow\ (c,s) \rightarrow\!* (\mathit{SKIP},t)$$

where $\Rightarrow$ is the big step and $\rightarrow$ the small step semantics.

5.2 Compiler

We compile to a simple stack machine with the following instructions:

$$\begin{aligned}\textbf{datatype}\ \mathit{instr} = {} & \mathit{LOADI\ int} \mid \mathit{LOAD\ vname} \mid \mathit{STORE\ vname} \mid \mathit{ADD} \\ & \mid \mathit{JMP\ int} \mid \mathit{JMPLESS\ int} \mid \mathit{JMPGE\ int}\end{aligned}$$

All jumps are relative. The compilation function *ccomp* is defined by recursion over the syntax. We prove that it preserves the semantics:

$$\mathit{ccomp}\ c \vdash (\mathit{0},\ s,\ \mathit{stk}) \rightarrow\!*\ (\mathit{isize}\ (\mathit{ccomp}\ c),\ t,\ \mathit{stk}) \ \longleftrightarrow\ (c,\ s) \Rightarrow t$$

The left-hand side describes the execution of the stack machine.

We have intentionally refrained from considering infinite executions as well. Leroy's elegant treatment [7] opens a whole new can of worms, coinductive definitions, which Isabelle knows about, but the students do not.

5.3 Typed IMP

We modify IMP by allowing both integer and real variables and values. There are no coercions and the semantics gets stuck when trying to add an integer and a real value. A type system for expressions is introduced and it is shown that the type systems fits the small step semantics, i.e. that well typed programs enjoy progress and preservation.

5.4 Static Analyses

We consider two iteration-free static analyses: definite assignment analysis as in Java and live variable analysis. Iterative analyses are considered later in the context of abstract interpretation.

Definite assignment analysis is defined as an inductive predicate D of type $\mathit{vname\ set} \Rightarrow \mathit{com} \Rightarrow \mathit{vname\ set} \Rightarrow \mathit{bool}$ that resembles a Hoare triple: the two sets represent the set of variables definitely initialized before and after the command. Soundness of the analysis w.r.t. a semantics that detects access of uninitialized variables is proved.

Live variable analysis is performed by a recursive function L that computes the set of variables live before a command given those that are live after the command. As mentioned above, the analysis is not iterative, it trades precision for efficiency:

$$L\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X = \mathit{vars}\ b \cup X \cup L\ c\ X$$

We also define a recursive optimization function *bury* that turns all assignments to dead variables into *SKIP*. Here are two of the defining equations:

$$\begin{aligned}&\mathit{bury}\ (x ::= a)\ X = (\mathsf{if}\ x \in X\ \mathsf{then}\ x ::= a\ \mathsf{else}\ \mathit{SKIP}) \\ &\mathit{bury}\ (c_1;\ c_2)\ X = \mathit{bury}\ c_1\ (L\ c_2\ X);\ \mathit{bury}\ c_2\ X\end{aligned}$$

We show that *bury* is sound: the big step transitions of c and *bury* c agree if the states are compared w.r.t. live variables only.

5.5 Security Type Systems

As a second and non-standard example of a type system we consider two versions of the Volpano-Smith-Irvine [20] security type system. First an executable one (following Section 3.6):

$$l \vdash \mathit{SKIP} \qquad \frac{\mathit{sec\text{-}aexp}\ a \le \mathit{sec}\ x \qquad l \le \mathit{sec}\ x}{l \vdash x ::= a} \qquad \frac{l \vdash c_1 \qquad l \vdash c_2}{l \vdash c_1;\ c_2}$$

$$\frac{\mathit{max}\ (\mathit{sec\text{-}bexp}\ b)\ l \vdash c_1 \qquad \mathit{max}\ (\mathit{sec\text{-}bexp}\ b)\ l \vdash c_2}{l \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2} \qquad \frac{\mathit{max}\ (\mathit{sec\text{-}bexp}\ b)\ l \vdash c}{l \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

And then the standard one based on a subsumption rule:

$$\frac{\mathit{sec\text{-}bexp}\ b \le l \qquad l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2} \qquad \frac{\mathit{sec\text{-}bexp}\ b \le l \qquad l \vdash' c}{l \vdash' \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

$$\frac{l \vdash' c \qquad l' \le l}{l' \vdash' c}$$

The first three rules for $\vdash'$ agree with the ones for $\vdash$ and have been omitted. We prove non-interference and the equivalence of the two systems.

5.6 Hoare Logic

We consider the standard partial correctness system and model assertions semantically as predicates on states. Soundness and completeness are proved. A verification condition generator (assuming loop-annotated programs) is defined and its soundness and completeness is proved. The details can already be found in the "Mechanized Semantics Textbook" [11].

5.7 Abstract Interpretation

We develop a generic abstract interpreter for IMP commands annotated with abstract states. Every command, except sequential composition, is annotated with the abstract state P after the command. The syntax is $\mathit{SKIP}\ \{P\}$, $x ::= a\ \{P\}$, $\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2\ \{P\}$ and $\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}$. The post-states P refer to the very end of each command, not to the end of the *ELSE* branch or the loop body. The I in $\{I\}\ \mathit{WHILE}\ b\ \mathit{DO}\ c\ \{P\}$ is the loop invariant. Starting from a program where all annotations are $\bot$ (annotations are in fact lifted abstract states, i.e. either $\mathit{Up}\ S$ or $\bot$), the abstract interpreter iterates a step function that maps annotated commands to annotated commands, changing only the annotations. Each step corresponds to the synchronous execution of one computation step at all points in the command. This corresponds to a Jacobi iteration on the corresponding dataflow equations.

Rather than go into the technical details, we explain the abstract interpreter by means of a worked example, interval analysis. An abstract state is a list of pairs (*x*,*ivl*) of variable names *x* and intervals *ivl*. An interval is of the form {*i*...*j*} where *i* and *j* are integers. Infinite lower or upper bounds are simply dropped. For example, {*1*...} is the set of all positive integers. We consider the iterated application of the step function to this program:

```
''x'' ::= N 7 {⊥};
{⊥}
WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 1) {⊥}
{⊥}
```

The first iteration tells us that *x* has value *7* after the assignment:

```
''x'' ::= N 7 {Up [(''x'', {7...7})]};
{⊥}
WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 1) {⊥}
{⊥}
```

The next step merely initializes the invariant:

```
''x'' ::= N 7 {Up [(''x'', {7...7})]};
{Up [(''x'', {7...7})]}
WHILE Less (V ''x'') (N 100) DO ''x'' ::= Plus (V ''x'') (N 1) {⊥}
{⊥}
```

One more step propagates the invariant to the end of the loop body:

```
''x'' ::= N 7 {Up [(''x'', {7...7})]};
{Up [(''x'', {7...7})]}
WHILE Less (V ''x'') (N 100)
DO ''x'' ::= Plus (V ''x'') (N 1) {Up [(''x'', {8...8})]}
{⊥}
```

In the next step, an ordinary join of {*7*...*7*} and {*8*...*8*} would result in the new invariant {*7*...*8*} and it would take many iterations until the actual invariant {*7*...*100*} is reached. Therefore we extend the abstract interpreter with widening and obtain the new invariant {*7*...} instead:

```
''x'' ::= N 7 {Up [(''x'', {7...7})]};
{Up [(''x'', {7...})]}
WHILE Less (V ''x'') (N 100)
DO ''x'' ::= Plus (V ''x'') (N 1) {Up [(''x'', {8...8})]}
{⊥}
```

For simplicity, widening is used at any point, not just for invariants. This means that when {*7*...} is pushed through the loop body it is first restricted to {*7*..*99*}, then incremented to {*8*...*100*}) and then widened with the previous {*8*...*8*} to {*8*...}. The post-state {*100*...} is obtained by a backwards analysis of the loop condition:

```
″x″ ::= N 7 {Up [(″x″, {7...7})]};
{Up [(″x″, {7...})]}
WHILE Less (V ″x″) (N 100)
DO ″x″ ::= Plus (V ″x″) (N 1) {Up [(″x″, {8...})]}
{Up [(″x″, {100...})]}
```

This is a post-fixed point and it is time to improve the result with narrowing. This time the post-state of the loop body is the result of narrowing {*8..100*} with the old {*8...*}, which yields {*8...100*}.

```
″x″ ::= N 7 {Up [(″x″, {7...7})]};
{Up [(″x″, {7...})]}
WHILE Less (V ″x″) (N 100)
DO ″x″ ::= Plus (V ″x″) (N 1) {Up [(″x″, {8...100})]}
{Up [(″x″, {100...})]}
```

Next time {*8...100*} narrows the invariant {*7...*} to {*7...100*}:

```
″x″ ::= N 7 {Up [(″x″, {7...7})]};
{Up [(″x″, {7...100})]}
WHILE Less (V ″x″) (N 100)
DO ″x″ ::= Plus (V ″x″) (N 1) {Up [(″x″, {8...100})]}
{Up [(″x″, {100...})]}
```

Backwards analysis of the loop condition turns the invariant into the post-state {*100...100*}:

```
″x″ ::= N 7 {Up [(″x″, {7...7})]};
{Up [(″x″, {7...100})]}
WHILE Less (V ″x″) (N 100)
DO ″x″ ::= Plus (V ″x″) (N 1) {Up [(″x″, {8...100})]}
{Up [(″x″, {100...100})]}
```

We have reached a fixed point and terminate.

The main advantage of this approach to abstract interpretation is its extreme concreteness and intuitiveness: the above example snapshots are the direct results of running the iterated step function. The abstract interpretation process is truly animated. In the presenece of widening/narrowing, the above iteration strategy is not optimal as Cachera and Pichardie [5] point out. They formalize a *denotational* abstract interpreter that is more precise (at least for the given examples, and possibly in general). From a teaching perspective we prefer the annotated commands approach because the student can see the results at intermediate stages by iterating the step function; a denotational approach yields the final result directly.

6 Conclusion

We have only taught the course 1 1/2 times so far. Hence it is difficult to draw definitive conclusions. But the overall tendency is very positive:

- The students earned 88% of the possible homework points, an unusually high number, especially considering that much of the homework consisted of proofs. We strongly conjecture that this is due to the use of a proof assistant (see Section 3.1) combined with the fact that the homework accounted for 40% of the final grade. One attempt at cheating was discovered. With more students than the current 15–20, plagiarism would become more of an issue.
- In the final exam, the results were significantly above average, both compared to previous editions of the course and to other theory courses. However, a comparison of the different exams is difficult because of differences like oral versus written. The final exam complemented the homework in that the exam concentrated on the semantics material, did not involve the proof assistant, but required the students to give informal proof sketches.
- The overall written feedback from the course was positive. The only negative comments concerned the amount of time the students spent on their homework ("too time consuming"). Looking at the overall departmental course evaluation statistics, only the database course was more demanding in terms of the amount of time the students invested (according to their own estimate). Unfortunately, this reflects the state of the art of proof assistants.

In summary: there were no more LSD trip proofs, the students had mastered both formal and informal proofs[3] and had a better understanding of the semantics material. Teaching Isabelle required the first quarter of the semester. We believe that this initial investment did not just improve the students' understanding of the logical foundations, it also allowed us to cover the semantics material more quickly than normally because of the solid and uniform foundations that could be taken for granted.

I concur with Pierce's assessment that this form of semantics course is the way of the future. There are many other courses out there that use proof assistants in some form or another (for example, Leroy's summer school course `http://cristal.inria.fr/~xleroy/courses/Eugene-2011/`), but there are few published accounts. Exceptions are ACL2 and DrScheme based courses in functional programming and software engineering [15, 16, 19] where automatic program verification and refutation is the focus and less the interactive construction of proofs. It will be interesting to see how much proof assistants will have an impact on teaching beyond the usual suspects of programming languages, formal methods and logic.

Acknowledgments Gerwin Klein has been a long term collaborator on this project. Sascha Böhme, Alex Krauss, Brian Huffman and Peter Lammich helped to run this course. All of them helped to debug this paper.

[3] Of course we should not forget that many of the proofs in this area follow standard patterns.

References

1. Scott Aaronson. Teaching statement, 2007. `http://www.scottaaronson.com/teaching.pdf`.
2. Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 131–146. Springer-Verlag, 2009.
3. Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lect. Notes in Comp. Sci.*, pages 24–40. Springer-Verlag, 2002.
4. Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems (FroCoS 2011)*, volume 6989 of *Lect. Notes in Comp. Sci.*, pages 12–27. Springer-Verlag, 2011.
5. David Cachera and David Pichardie. A certified denotational abstract interpreter. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lect. Notes in Comp. Sci.*, pages 9–24. Springer-Verlag, 2010.
6. Gerwin Klein. Interactive proof: Applications to semantics. In *Proc. Summer School Marktoberdorf 2011*, 2012. To appear.
7. Xavier Leroy. Coinductive big-step operational semantics. In P. Sestoft, editor, *Programming Languages and Systems (ESOP 2006)*, volume 3924 of *Lect. Notes in Comp. Sci.*, pages 54–68. Springer-Verlag, 2006.
8. Adam Naumowicz and Artur Kornilowicz. A brief overview of Mizar. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 67–72. Springer-Verlag, 2009.
9. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley, 1992.
10. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications. An Appatizer*. Springer-Verlag, 2007.
11. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
12. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
13. Tobias Nipkow. Interactive proof: Introduction to Isabelle/HOL. In *Proc. Summer School Marktoberdorf 2011*, 2012. To appear.
14. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002.
15. Rex Page. Engineering software correctness. *J. Functional Programming*, 17(6):675–686, 2007.
16. Rex Page, Carl Eastlund, and Matthias Felleisen. Functional programming and theorem proving for undergraduates: A progress report. In *Proc. 2008 International Workshop on Functional and Declarative Programming in Education*, FDPE '08, pages 21–30. ACM, 2008.

17. Benjamin C. Pierce. Lambda, the ultimate TA: using a proof assistant to teach programming language foundations. In *Proc. 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 121–122. ACM, 2009.
18. Christian Urban. Nominal techniques in Isabelle/HOL. *J. Automated Reasoning*, 40:327–356, 2008.
19. Dale Vaillancourt, Rex Page, and Matthias Felleisen. ACL2 in DrScheme. In P. Manolios and M. Wilding, editors, *Proc. Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 107–116. ACM, 2006.
20. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.
21. Markus Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.
22. Markus Wenzel and Freek Wiedijk. A comparison of Mizar and Isar. *J. Automated Reasoning*, pages 389–411, 2002.
23. Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.