Interactive Teaching of Programming Language Theory with a Proof Assistant

BERECZKY Péter, DONKÓ István,
HORPÁCSI Dániel, KAPOSI Ambrus, NÉMETH Dávid János

Abstract. Teaching of programming language theory has a long track record at ELTE Faculty of Informatics. Traditionally, formal semantics and type systems of programming languages, similarly to other theory-oriented subjects, were taught with the pen and paper method. However, modern proof assistants call for replacing this old-fashioned way of teaching with novel and interactive methods that bring deeper understanding, provide better learning experience and build technical skills in applying formal methods. The authors have launched practice classes for two programming language theory subjects and carefully developed course material based on executable and verifiable definitions formalised in the Coq proof assistant. In this paper, we share our experiences regarding the design and implementation of the new material, we outline the pros and cons of using a proof assistant in the courses, and we describe how the presented method may be adapted to other courses.

Keywords: formal semantics, type systems, proof assistant, Coq, interactive teaching

1. Introduction

Teaching mathematically precise formalisation techniques is a key part of university level Computer Science education. The courses titled *Formal Semantics* and *Type Systems for Programming Languages* are compulsory in the Software Technology specialisation of the Computer Science master program at Eötvös Loránd University (ELTE). The aim of these courses is that students acquire the skills necessary to apply mathematical methods when describing programming languages. During these courses the students have to digest and obtain intuition for a large number of abstract concepts and formal notations. This is especially hard for students with less mathematical affinity. By sort of "wrapping" the raw mathematical contents into detailed, practical examples, we can help the students grasp the constructions and obtain understanding. What helps even more is giving the students tools to freely experiment with which gives immediate feedback. To achieve this, we started employing computer proof assistant systems in our practical sessions. The current article describes the experiences gained from teaching with this new methodology.

2. Proof assistants in education

Proof assistants (interactive theorem provers) are software which give the user the ability to:

- Define detailed abstract models
- Construct statements over previously defined models
- Interactively prove statements using mathematical methods, the correctness of which can be checked automatically by computers

Several such systems were considered during the planning phase of our courses, comparing them based on previous experience in the literature regarding teaching and their compatibility with the previously outlined goals.

The Haskell programming language [6] is strictly speaking not an interactive theorem prover, but its purely functional nature and simple syntax makes it suitable for modelling programming languages. The abstract syntax can be represented with algebraic data types and the semantics can be described using executable functions in a denotational style. Haskell was previously used to

teach programming language semantics [11], but as it lacks dedicated theorem proving functionality, we decided against it.

Agda [1] is another functional programming language with a type system stronger than Haskell's. It supports dependent types [14], that is, types that depend on values. This strength allows representing mathematical statements as types, and constructive proofs as executable functions that implement the given type. This is usually called the propositions as types principle or Curry-Howard isomorphism [15]. Agda was also used before as a tool to teach programming language theory [10]. It uses the same notation for proofs and programs which makes it simple but confusing at the same time because paper-based proofs look very different from Agda proofs.

The Isabelle/HOL theorem prover [7] meets these requirements and its educational potential has also been demonstrated [2], but its type system does not support dependent types. This makes it more verbose, by requiring the user to manually handle information in certain cases that could otherwise be encoded into the types themselves.

We chose to use the Coq proof assistant [5], as it has a dependently typed specification language (*Gallina*) that is similar to Agda in expressive power, but it also comes with a separate tactic system that can be used to construct imperative style proofs which are similar to pen and paper proofs. Its widespread use in education [3], detailed auxiliary materials [9] and active support [4] have further strengthened its leading position.

3. The Formal Semantics course

In contrast to natural languages, programming languages are artificially created by humans for the purpose of efficiently controlling computers. Because of this, the meaning of their statements needs to be clear without any chance of ambiguity, so they can be interpreted by a computer. *Formal Semantics* deals with the task of describing the behaviour of programming languages and through this process defining the meaning of programs written in them.

The course has always played an important role in the curriculum of the Computer Science master programs at ELTE. It provides insight into the mathematical methods involved in formalisation of the meaning of programming languages. Different ways of describing the syntax are described, as well as static and dynamic semantics, including operational, denotational and axiomatically given semantics. Well known real world (mostly imperative) programming languages serve as examples in the formalisation process. This makes it easier to connect this new information with previous studies and helps deepening the understanding of the already acquired knowledge.

3.1 Aim of the course

The course has multiple goals, the most important of which is to popularise the formal definition of programming languages by clarifying the role of this process in the understanding, comparison and analysis of programs, as these are the key steps in proving correctness of a program. Students create their own mathematically precise descriptions of several programming language constructs they are already familiar with, e.g. branches, loops or exception handling. When doing this they need to carefully examine the behaviour of these constructs and thus gain a better understanding of their interactions, limits, the similarities and differences between their different implementations in various programming languages.

Throughout the classes several concepts from previous courses (such as syntax definitions from the *Compilers* course) are used which helps in refreshing, reinforcing and expanding their already established knowledge. We introduce methods for defining operational semantics (small-step and big-step) through the notions of configurations and inductively defined transition relations between them. Denotational semantics is explained using executable semantic functions and the principle of compositionality.

Furthermore, many of the mathematical concepts that the students encounter for the purpose of applying them in formal semantics are from set theory or algebra, thus applicable in their future studies. Examples are inductive definitions, pattern matching, induction, or parts of fixed-point theory.

3.2 Previous renditions of the course

The initial version of the course in the more mathematically focused program was made up of lectures and practical lessons (both 90 minutes each week for one semester), during which students could practice the use of the mathematics discussed in the lectures, but no interactive techniques were employed, the classes were based on the pen and paper method. From 2008 onwards the practical sessions were removed, only the lectures remained part of the course (with the same amount of time). This affected the effectiveness and popularity of the course negatively, as students had a hard time fully understanding the abstract concepts without adequate practice.

3.3 The new practical lab classes

In 2018 the course was renewed by including 90 minutes of practical session per week in addition to the 90-minute lectures. The syllabus had to be constructed in a way which was suitable for all students in their second semester of their master's studies. The instructors have agreed that giving the students a way in which they can experiment themselves is highly beneficial and being able to do so interactively through a computer is even better. As the goal was to not only formalise the definitions, but also prove theorems about them, a system with such capabilities was needed.

3.4 Main principles

The primary task of the practical sessions is to help the students understand the lecture material. This means formalising definitions, expressions, programs, theorems and examples. Unfortunately, most of the students are not familiar with proof assistants, therefore they need to learn how to use Coq without losing focus: this course should be on formal semantics and not the technical details of Coq's implementation. Our experience was that the following educational principles were helpful in this task.

1. **Understanding at the lectures, coding at the practical sessions.** The lecture material is presented sometimes through very complex examples in order to model real programming language constructions closely. These examples are discussed in detail in the lecture, but their machine-checked formalisation is too difficult for a beginner proof assistant user to implement. We could simply share the source code of such formalisations with the students (as it was the case with source code for executable semantics in previous renditions of the course), but one goal of the new practical sessions is to involve the students even more. In the practical sessions most of the time is spent on formalising

informal or half-formal definitions from scratch or using the material of the previous lessons. The students implement the definitions and prove theorems independently, rather than reading or modifying other existing formalisations. This way the time is not spent on understanding and discussing complex problems, but on practicing formalisation using smaller examples. Therefore, the sessions are focused on programming language semantics while at the same time the students gain experience in using Coq.

2. **First understand, then code.** The previous principle should not imply that students should blindly do coding before understanding the definitions, theorems and proofs. In fact, the practical session has the additional task to separate the understanding of the theoretical materials from the formalisation in Coq. During these sessions, the students try to solve small problems in complete detail, but only after they have understood the theory in detail. For this reason, whiteboard and paper is used when it is necessary, but these "analog" exercises get less emphasis than during the exercise sessions for other mathematical courses.
3. **Graduality in familiarising with the proof assistant.** Coq allows its users to create formalisations conveniently with a large number of language features. However, if we aimed to teach the students how to write idiomatic Coq code, we would end up not teaching formal semantics, but rather Coq through formal semantics examples. To keep the focus on the topic of the course, the language elements and functions of the theorem prover are introduced step-by-step; three-four elements (commands, tactics, etc.) per practical session. This style is similar to that of the book Programming Language Foundations [9]. The result is that students gain knowledge of the concepts and language of Coq progressively without getting lost in its details.
4. **Graduality in the complexity of the theorems.** The students usually neither have sufficient knowledge of theorem provers, nor of the process of theorem proving. Because of this reason, some proof theory has to be taught at the Formal Semantics course, while discussing the method of formalising proofs. In order to keep the focus and enthusiasm of the students, the complexity of proofs should be increased with caution. According to previous experience, the master's students have no problems with formalising simple functions with case distinction, however, that is not true for the transformation of derivation rules to inductive definitions. With adequate preparation we can maintain a continuous sense of achievement during the semester, which makes the usage of proof assistants a positive experience. This can be an important milestone in the students' professional development.
5. **Continuous work and short tests.** The requirements of the course were planned so that the students have to study every week. There is an optional homework assignment every week and each practical session starts with a short assignment similar to the homework. The results of these short tests determine the final grade for the practical sessions (students obtain a grade separately for the lecture). Every assigned task has to be solved in Coq to accelerate the acquisition of the proof assistant. The assignments are submitted in an online e-learning platform and the students get immediate feedback on the correctness of their submissions. This setup helps the students stay motivated throughout the semester.

3.5 Syllabus

The syllabus of the practical sessions is primarily based on the lecture materials and the aforementioned book [9]. Fortunately, the book gives an excellent basis for the swift introduction

to Coq, while the formal semantics examples discussed in it are often very similar to the ones presented during the lectures, because both sources deal with the concepts of imperative programming. The syllabus of the practice was assembled to follow and formalise the materials of the book during the first half of the semester and the materials of the lectures during the second half.

Below we describe the contents of the practical sessions for each week. To illustrate the increasing complexity of the material, we include four example Coq source codes snippets. We also list the usual informal proofs for comparison (although not always exactly these theorems are discussed during the lectures).

1. Introduction of the proof assistant by formalising bool type, then defining functions (e.g. and, or, not) followed by lemmas and theorems and their proofs (e.g. commutativity of and) by case distinction.
2. Inductive definition of natural numbers. Introduction of structural recursion (and pattern matching) by defining recursive functions (e.g. addition) and structural induction to prove properties about these functions in the form of lemmas, theorems.

```
Inductive Nat : Type :=
| O
| S : Nat -> Nat.

Fixpoint plusn (n m : Nat) : Nat :=
match n with
| O => m
| S n' => S (plusn n' m)
end.

Notation "n + m" := (plusn n m)
 (left associativity, at level 50).

Theorem plusn_rid : forall n : Nat,
n + O = n.
Proof.
  intros.
  induction n.
  * simpl. reflexivity.
  * simpl. rewrite IHn. reflexivity.
Qed.
```

Assume that natural numbers are defined based on the Peano-axioms with the 0 constant and the S successor function.

We define the addition in the following way, and denote with "+":

$$add(n,m) = \begin{cases} m & n = 0 \\ add(n',m) & \exists n' : n = S(n') \end{cases}$$

Theorem: $n + 0 = n$.

This statement can be proven by induction on n.

- Base case $= 0 : 0 + 0 = 0$ is true according to the definition of *add*.
- Induction hypothesis: $n + 0 = n$.

The statement to prove: $S(n) + 0 = S(n)$.
By the definition of *add*: $S(n) + 0 = S(n + 0)$.
By the induction hypothesis: $S(n + 0) = S(n)$.

3. Formalisation of binary trees followed by the expression language syntax with inductive definitions (deep embedding). Static semantics: mappings, functions with the domain of trees or expressions (e.g. the number of leaves in a tree, the number of literals, operations in one expression). Formalisation of the denotational semantics for the given expression language. Simple inductive proofs about the trees and expressions.
4. Transformation of expressions to equivalent expressions (optimisation mappings). Proofs about the meaning preservation of these transformations in the denotational semantics.
5. Introduction of states (variable environment) and the extension of the expression syntax and semantics with variables followed by lemmas and proofs about the behaviour of states.

```
Inductive aexp : Type :=
| ALit (n : nat)
| AVar (x : ident)
| APlus (a1 a2 : aexp).

Definition state : Type :=
```

We define the syntax of expressions with BNF ("n" denotes a natural number, "x" a string):

$$a \in Aexp ::= n \mid x \mid a_1 + a_2$$

```
ident -> nat.

Fixpoint aeval (a: aexp) (s : state) :
nat :=
match a with
| ALit n => n
| AVar x => s x
| APlus a1 a2 =>
aeval a1 s + aeval a2 s
end.

Definition update (s:state) (x:ident)
(n:nat) : state :=
fun y => if eqb x y
then n
else s y.

Lemma update_onlyx:
forall s:state, forall x x':ident,
forall n:nat,
~(x = x') -> (update s x n) x' = s x'.
Proof.
  intros. unfold update.
  rewrite <- eqb_neq in H.
  rewrite H. reflexivity.
Qed.
```

A state is a function from variable identifiers to natural numbers:

$$s \in State = ident \to N$$

The denotational semantics is a recursive function on expressions:

$$A: Aexp \to (State \to N)$$

$$A[\![n]\!]s = n$$

$$A[\![x]\!]s = s(x)$$

$$A[\![a_1 + a_2]\!]s = A[\![a_1]\!]s + A[\![a_2]\!]s$$

We will use $s[y \to n]$ to denote that s' state which is the same as s except $s'(y) = n$.

Theorem: for any s state, if $x \neq x'$ then $s[x \to n](x') = s(x')$.

The proof is simple. The $s[y \to n]$ and s are different only in what values that are mapped to x. So, for any x' that is different from x, these values will be the same.

6. Static analysis: free and bound variables, function and inductive definition (relation) about an expression being closed followed by lemmas and proofs about closed expressions.
7. Formalisation of small-step semantics for expressions. Practicing the formalisation of the inference rules to inductive definitions. Evaluation of example expressions in the presented semantics.
8. Formalisation of big-step semantics for expressions. Proof of equivalence with other (denotational or small-step) semantics.

```
Reserved Notation "c -=> c'" (at
level 50).

Inductive eval_bigstep : aexp * state
-> nat -> Prop :=
| eval_lit n s:
  (ALit n, s) -=> n
| eval_var x s:
  (AVar x, s) -=> s x
| eval_plus a1 a2 n m s:
  (a1, s) -=> n ->
  (a2, s) -=> m
 ->
  (APlus a1 a2, s) -=> (n + m)

where "c -=> c'" :=
  (eval_bigstep c c').

Theorem denot_iff_bigstep :
  forall a:aexp,
  forall s:state,
```

The big-step semantics can be described with inference rules between configurations of an expression and a state and a natural number:

$$\frac{}{\langle n, s\rangle \Rightarrow n}$$

$$\frac{}{\langle x, s\rangle \Rightarrow s(x)}$$

$$\frac{\langle a_1, s\rangle \Rightarrow n \quad \langle a_1, s\rangle \Rightarrow m}{\langle a_1 + a_2, s\rangle \Rightarrow n + m}$$

Theorem: The equivalence of the big-step and denotational semantics, i.e. for every expression a and state s and natural number n, $A[\![a]\!]s = n \leftrightarrow \langle a, s\rangle \Rightarrow n$.

We provide proof for the forward direction in this paper. This way, the hypothesis is$A[\![a]\!]s = n$.We use induction by a.

- We must prove that $\langle n', s\rangle \Rightarrow n$, i.e. $n = n'$. But according to the hypothesis, $A[\![n']\!]s = n$, that is only possible when $n = n'$.

<table>
<tr>
<td>

```
  forall n:nat,
  aeval a s = n <-> (a,s) -=> n.
Proof.
  split.
  * generalize dependent n.
    induction a.
 - intros. subst. apply eval_lit.
    - intros. subst. apply eval_var.
    - simpl. intros. subst. apply
eval_plus.
      + apply IHa1. reflexivity.
      + apply IHa2. reflexivity.
  * intros. generalize dependent n.
    induction a.
    - simpl. intros. inversion H.
      reflexivity.
    - simpl. intros. inversion H.
      reflexivity.
    - intros. simpl. inversion H.
      rewrite (IHa1 n0 H4).
      rewrite (IHa2 m H5).
      reflexivity.
Qed.
```

</td>
<td>

- We must prove that $\langle x, s\rangle \Rightarrow n$, i.e. $s(x) = n$. According to the hypothesis $A[\![x]\!]s = n$, we know that $s(x) = n$, because of the definition of the denotational semantics.
- Induction hypotheses:
  - $\forall n: A[\![a_1]\!]s = n \leftrightarrow \langle a_1, s\rangle \Rightarrow n$
  - $\forall n: A[\![a_2]\!]s = n \leftrightarrow \langle a_2, s\rangle \Rightarrow n$

According to the hypothesis $A[\![a_1 + a_2]\!]s = n$, that means $A[\![a_1]\!]s + A[\![a_2]\!]s = n$. To $\langle a_1 + a_2, s\rangle \Rightarrow n$ it is sufficient to prove that $\langle a_1, s\rangle \Rightarrow A[\![a_1]\!]s$ and $\langle a_2, s\rangle \Rightarrow A[\![a_2]\!]s$ which are exactly the induction hypotheses with choosing $n = A[\![a_1]\!]s$ in the first case, and $n = A[\![a_2]\!]s$ in the second.

</td>
</tr>
</table>

9. Introduction of imperative programming statements. Formalisation of their syntax and denotational semantics. Fixpoint theory, the question of termination.
10. Extension of the previous big-step semantics with statements followed by example program evaluation proofs.
11. Formalisation of additional examples, the extension of the semantics with other statements (e.g. counting loop as an inductive rule or as syntactic sugar). Proofs about the equivalence between big-step and denotational semantics.
12. The equivalence of program patterns (e.g. while loop can be unfolded to a conditional statement containing a similar loop).

<table>
<tr>
<td>

```
Lemma while_unfold (b:bexp) (s:stmt)
(st st':state):
  (SWhile b s, st) -=> st' <->
  (SIf b (SSeq s (SWhile b s))
          SSkip, st) -=> st'.
Proof.
  split.
  * intros. inversion H.
    - subst.
      apply eval_if_true.
      apply (eval_seq st'0 _ _ _ _
H3 H5).
      exact H6.
    - apply eval_if_false.
      + apply eval_skip.
      + exact H4.
  * intros. inversion H.
    - subst. inversion H5. subst.
      apply (eval_while_true st'0).
      + exact H4.
      + exact H7.
      + exact H6.
```

</td>
<td>

Theorem:
$\langle \mathit{while}\ b\ \mathit{do}\ S, s\rangle \Rightarrow s' \leftrightarrow$
$\langle \mathit{if}\ b\ \mathit{then}\ S; \mathit{while}\ b\ \mathit{do}\ S\ \mathit{else}\ \mathit{skip}, s\rangle \Rightarrow s'$.

The hypothesis $\langle \mathit{while}\ b\ \mathit{do}\ S, s\rangle \Rightarrow s'$ could only be gotten by two different ways: either b evaluates to true or false.

If b evaluates to false, then according to the semantics of the loop, the original state s will be the result. So, to prove the original statement, it is sufficient to prove (because of the condition of the **if** statement evaluated to false) that $\langle \mathit{skip}, s\rangle \Rightarrow s$ which is exactly the semantics of skip.

If b evaluates to true, then according to the semantics of the loop, we get two new hypotheses: $\langle S, s\rangle \Rightarrow s_1$ and $\langle \mathit{while}\ b\ \mathit{do}\ S, s_1\rangle \Rightarrow s'$. In this case it is sufficient to prove (because the condition of the **if** statement evaluated to true) that $\langle S; \mathit{while}\ b\ \mathit{do}\ S, s\rangle \Rightarrow s'$. In order to prove this statement, the inference rule of the sequence can be used. We need to provide an intermediate s_2 state,

</td>
</tr>
</table>

`- subst. inversion H5.`
`apply eval_while_false.`
`rewrite <- H0. exact H6.`
`Qed.`
	that conforms the next two statements: $\langle S, s\rangle \Rightarrow s_2$ and $\langle while\ b\ do\ S, s_2\rangle \Rightarrow s'$. Fortunately, s_1 is such a state.

13. Extraction of correct Coq code to Haskell. Formal semantics in the practice, industry.

Because of the structure of this step-by-step material, the code is usually reused from previous session (as is the case for the denotational semantics or the syntax of expressions in the example codes above). In order not to spend time on redefining these always from scratch, the necessary source code is shared with the students before the session. In addition, it is also useful to make the code discussed in the lesson accessible which can be used by students later when they solve the homework or practice for the next test.

3.6 Deep understanding of complex concepts

As mentioned before, there are concepts and methods that are challenging to understand, even for master's students. The practical sessions have a very important role in this perspective, which is to introduce these complex, abstract concepts through small, simple examples. Therefore, during the semester these are practiced on gradually larger and harder examples. According to our previous experience, the teaching of the following concepts became easier due to the practical sessions:

- *Derivation rules (inference rules) and derivations*: It is not always clear for the students why do we formalise the operational semantics in a concrete way; what can be written over and under the line (which separates the premises and conclusion) of the inference rule. The students are forced by the system to follow the rules of the formalism given by inductive types in Coq. As a consequence, the students grasp faster and easier why, how and where derivation rules can be applied. Probably this is also due to the pattern matching mechanism in Coq, which is familiar to students who previously studied functional programming (which is a compulsory course in our bachelor's programme).
- *Structural induction*: The students learned about mathematical induction (for natural numbers), but they are not familiar with the concept of structural induction for arbitrary inductively defined sets. So, the scheme used for inductive proofs is not always understandable for them in the case of structurally complex types; which hypotheses should be used, which statements are not correct implying an incorrect assumption, what should the induction be based on, etc. Usually, paper-based proofs are more readable, however, they can be faulty or only partial. On the other hand, the proving process in Coq is interactive (using its integrated development environment, *CoqIde*), so that the students are led by the proof assistant through the process step-by-step. In every step, Coq reports the current hypotheses and the statements to be proven, and it does not allow to take faulty steps. Last but not least, each proof goal has to be proven, including trivialities.
- *Compositionality*: The principle of compositionality is usually just learned word by word by the student, but they do not understand its essence and its potential in practical applications. However, when proving a statement by structural induction on expressions the proof assistant shows clearly, that the induction hypotheses are the same as the statements to be proven about complex structures thanks to the compositionality. Similarly, Coq's termination checker accepts compositional recursive definitions but rejects those where recursive calls are not on structurally smaller arguments.

3.7 Results

The effectiveness of the new practical course can be measured objectively by looking at the grades received at the end of the semester. Here we only evaluate the exams based on the syllabus of the lectures as these had the same requirements as in the previous years. This was the first year of the practical sessions, so there is not enough data to draw far-reaching conclusions.

The oral exam aims to measure the lexical knowledge as well as the understanding of the material. Because of the large number of students there is also a written exam which helps in filtering out those who are not sufficiently prepared and also gives a chance to those who are not really interested in the subject to get a passing grade. If someone successfully completes that, they can choose to continue with the oral exam to obtain a better grade.

3.7.1 Exam results

The reintroduction of the practical course has clearly improved the results of the exams. This is most likely due to the students not only memorizing the theoretical subject matter, but actually learning and understanding it more. In the following we compare the results of the 2018 Spring (57 students) and 2019 Spring (61 students) semesters.

- **Improvement of the average mark.** In Hungary, the grades range from 1 (failed) to 5 (excellent). When looking at all the participants of the exam, the average of all received marks raised from 2.6 to 3.14 without significant changes to the written pre-exam and having the same people conduct the exams (see Figure 1).

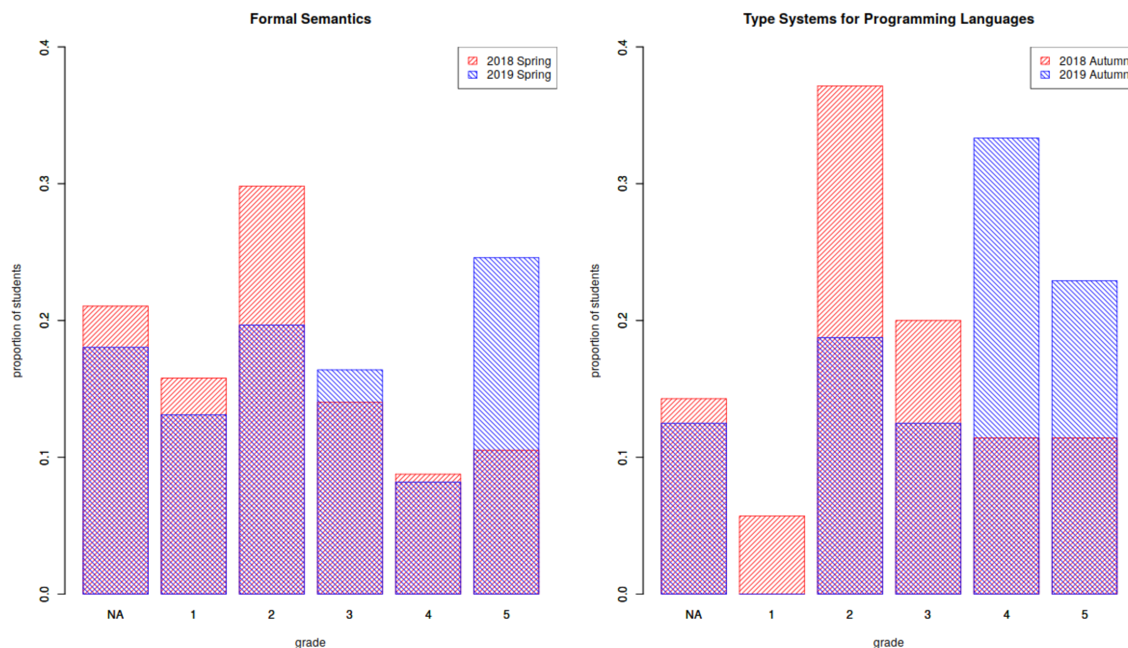

Figure 1: Exam results in the two investigated courses comparing the semesters without (red) and with (blue) practical sessions.

- **Increased participation in the oral exam.** Although students have the option to leave with a passing grade (2) after a successful written pre-exam, this year the ratio of those who chose to take the opportunity to continue increased from 33% to 48%. This shows that

they were more confident in their knowledge and almost half of them aimed for a higher mark instead of just around a third of them as the year before.

- **Improvement of the average mark at the oral exam.** The oral exam requires students to demonstrate their detailed knowledge and understanding of a specific part of the subject. This year they performed much better when asked to explain their assigned topic, which further confirms that they have internalized it more. The average mark of those who took the option of the oral examination raised from 3.9 to 4.2.

3.7.2 Subjective results

From the perspective of the teachers, the improvement was apparent when looking at the fluency of the exams and the confidence of the students. They seemed to understand the concepts, the steps and the general structure of proofs better as well. The improvement seen numerically in the results was perceivable subjectively too.

Positive feedback has also arrived from the students. Thanks to the interactive methods in the practical sessions they approached the subject with more enthusiasm. In the long term, this approach seems to be capable of delivering usable knowledge to average students who would otherwise not be interested enough to invest time in understanding hard mathematical concepts on their own.

4. The Type Systems for Programming Languages course

4.1 Aim and main principles of the course

In the 2019 Autumn semester the Coq proof assistant was also utilised for teaching practical sessions in another course titled *Type Systems for Programming Languages*. This is a third semester master's course which similarly to Formal Semantics, did not have practical sessions before. The structure and grading of these classes were very similar to the format of Formal Semantics outlined above. There was a small assignment every week that helped practice the new language constructs, notions and concepts introduced during the preceding session. These were followed by a short test at the beginning of every session that was similar to that week's homework, so that those who spent time on it at home had a significant advantage and thus were motivated to do so. In later phases of the course the students were encouraged to prepare their own reusable tactics at home and bring their code to the classes. This opportunity was taken only by a few of them, but those who chose to do so clearly showed that trying to look at the problem from a general perspective instead of only concentrating on solving one specific proof helped them gain a deeper understanding of the recurring patterns often used in proofs. The results of these small tests determined the final grade for the practical course. At the end of the semester every student also got a bigger homework in the form of a formalised language with a type system and had to prove a theorem for it. The languages and theorems were randomly assigned in a way that ensured a unique task for each student.

4.2 Syllabus

Instead of the imperative constructs discussed in *Formal Semantics* (assignment, branching, loops, etc.), this course uses an expression language for demonstration purposes that is closer to the functional paradigm. There is an operational semantics defined for this language, and it is extended

with typing relations. The arc of the whole semester was defined by following Parts I—VII of Harper [12] and Hungarian lecture notes based on them [13]. The first few sessions were mostly rehearsing the basics of Coq through small illustrative examples with terms and types, such as a recursively defined type checker on simple inductively defined terms, because not all students encountered the language before. After this, basic inductive proofs were introduced, and the soundness and completeness of the previously defined type inference algorithm were proven. This was followed by creating an inductive typing relation and an inductive transition relation which opened the opportunity to discuss operational semantics and type systems separately as well as their interactions. Further into the course the notion of contexts and well-formedness were presented, and multiple important theorems were proven that connected the previous concepts, such as the *Unicity of Typing*, the *Substitution Lemma*, the *Lemma of Decomposition*, the *Theorem of Determinism*, the *Theorem of Progress* and the *Theorem of Type Preservation*. The following snippets illustrate a few smaller pieces of code, as the full sources would be unsuitable for incorporation into the article because of their lengths.

Partial definition of the typing relation:

```
Inductive TypeJudgement : Con -> Tm -> Ty -> Prop :=
[...]
  | TJ_plus {G : Con} {t t' : Tm} : a
      (G |- t : Nat) -> (G |- t' : Nat) ->
      (G |- (t + t') : Nat)
[...]
where "G |- tm : ty" := (TypeJudgement G tm ty).
```

$$\frac{\Gamma \vdash \mathrm{t} : \mathrm{Nat} \quad \Gamma \vdash \mathrm{t}' : \mathrm{Nat}}{\Gamma \vdash \mathrm{t} + \mathrm{t}' : \mathrm{Nat}}$$

Figure 2: Coq formalization (left) and traditional notation (right) of an inductive type judgment constructor definition stating that if two terms (`t` and `t'`) can both be typed as `Nat` in a certain context, their sum can be proven to be of type `Nat` as well in the same context.

Partial definition of the small-step transition judgment:

```
Inductive OneStepTransitionJudgement : Tm -> Tm -> Prop :=
| OSTJ_sum {n1 n2 n : nat} :
    ((n1 + n2)%nat = n) ->
    num n1 + num n2 |--> num n
[...]
| OSTJ_plus_left {t1 t1' t2 : Tm} :
    (t1 |--> t1') ->
    t1 + t2 |--> t1' + t2

| OSTJ_plus_right {t1 t2 t2' : Tm} :
    t1 val -> (t2 |--> t2') ->
    t1 + t2 |--> t1 + t2'
[...]
where "t |--> t'" := (OneStepTransitionJudgement t t').
```

$$\frac{n_1 + n_2 = n}{num\ n_1 + num\ n_2 \longmapsto \mathrm{num}\ n}$$

$$\frac{t_1 \longmapsto t_1'}{t_1 + t_2 \longmapsto t_1' + t_2}$$

$$\frac{t_1\ val \quad t_2 \longmapsto t_2'}{t_1 + t_2 \longmapsto t_1 + t_2'}$$

Figure 3: Coq formalization (left) and traditional notation (right) of an inductive small-step operational semantics definition concerning the evaluation of the addition of two terms.

Partial proof for the Theorem of Progress:

<table>
<tr>
<td>

```
Theorem progress {t : Tm} {A : Ty} :
  ( * |- t : A ) ->
  t val \/ (exists (t' : Tm), t |--> t').
Proof.
  intros. remember * as G. induction H.
[...]
  - destruct (IHTypeJudgement1 HeqG).
    + destruct (IHTypeJudgement2 HeqG).
      * pose (n1 := progress_helper_Nat H H1).
        inversion n1. rewrite H3.

        pose (n2 := progress_helper_Nat H0 H2).
        inversion n2. rewrite H4.

        right. eexists.
        refine (OSTJ_sum _). reflexivity.

      * inversion H2. right. eexists.
        exact (OSTJ_plus_right H1 H3).

    + inversion H1. right. eexists.
      exact (OSTJ_plus_left H2).
[...]
Qed.
```

</td>
<td>

Induction is initiated on `H`, the hypothesis claiming that the term `t` has type `A` in the empty context.

In case the typing judgment was constructed using `TJ_plus` (see Figure 1.) the inductive hypotheses state that the theorem holds for the left and right operands separately. There are two branches based on the first hypothesis:

+ If `t` is already a value we further split the proof based on the second hypothesis:
  * `t'` is already a value as well, in which case they must both be in the form of `num n`, which means that the sum rule can be applied
  * `t'` can be rewritten, which means that the rule that rewrites the right-hand side can be applied
+ In case `t` can be further rewritten we simply apply the rule that rewrites the left-hand side of the addition

</td>
</tr>
</table>

Figure 4: Coq formalization (left) and explanation (right) of a branch from the inductive proof of the *Theorem of Progress*. It states that if a term can be properly typed in a certain context, then it is either already a value, or can be further evaluated using the small-step operational semantics.

4.3 Results

This way of incorporating computers into the teaching had some advantages, like the ability to automatically grade assignments by just type checking them, but also had brought some disadvantages, such as the practical lesson being a lot slower and after a while lagging behind the lecture because of the strict nature of the interaction with a proof assistant that did not allow the students to progress further without getting all the details right. To alleviate this, the skeletons of files (that contained most of the definitions and theorems) were prepared for every class and the students only had to fill in the missing parts of the definitions and proofs themselves. Because of this some students were able to advance more quickly and get through the file before the end of the class. Creating some harder optional tasks for further exercise at the end of the files was a great success and resulted in the retention of the attention of the faster students as well.

The new practical sessions were introduced in the 2019 Autumn semester. Comparing the results of the 2018 Autumn (35 students) and 2019 Autumn (48 students) exams (see Figure 1), we observe that the average went up from 2.8 to 3.7. The style and difficulty of the exams were the same in the compared semesters (written exam in the exam period).

5. Coq in other courses

Apart from the already mentioned two courses, there are several other subjects that could potentially benefit from the introduction of computer-based proof verification in the topic of programming theory, logic and mathematics. In the last semester work on a formalisation of the syllabus of *Distributed Systems* begun by a student [8]. In the long term it seems convincing that the introduction of proof assistants spanning across several subjects could greatly improve the proficiency of the students in several fields and thus increase the quality of the Computer Science program. Other courses which might apply proof assistants include *Logics* and *Computability theory*.

6. The challenges of using theorem provers

Among the desirable traits of the proof assistants that facilitate the learning process there happen to be some that also pose pedagogical challenges. Some of these arise because of design choices made during the development of Coq, while some others are consequences of one fundamental difference between proofs written on paper and those that can be verified by computers.

For example, while functional programming is now a compulsory part of the bachelor's program, the syntax of Coq is still strange for most of the students, as it follows the OCaml style [16] with which most of them are not familiar. Showing them the same definition in multiple languages – in object-oriented alternatives as well – can help in overcoming this obstacle.

The aforementioned difference from the usual blackboard reasonings they are used to is that a computer can only check the correctness of a proof if every little technical detail is given for all the branches created by different cases that need to be dealt with, lots of which are often omitted in lectures and textbooks because they can easily be seen intuitively. Luckily this can be simplified by creating reusable tactics that are generic enough to cover the repetitive sections of proofs, thus reducing the verbosity and increasing the similarity to their respective paper-based counterparts.

7. Summary

In this article we showed how interactive practical sessions using computer proof assistants were launched for courses on programming language theory that had only been taught using pen and paper methods before. The motivation behind this was to support the abstract concepts in these courses with tangible examples that help understanding the material and lead to building skills in applying formal methods. Several interactive theorem provers were considered which support executable and verifiable definitions, but Coq proved to be outstanding with its separated specification and proof languages, expressive type system as well as the abundantly available literature.

We have identified guiding principles in applying interactive theorem proving in our master's classes, such as "First understand, then code", "Graduality" and "Continuous work and short tests". These ensure that the programming language theory classes are adequately augmented by the practical sessions and the students gain a better understanding of nontrivial concepts such as deduction, induction or compositionality. Introducing the new system naturally came with technical and pedagogical challenges. Students needed to familiarise themselves with machine-assisted theorem proving and the programming language of Coq, but this initial investment payed off by the end of the semester.

The effectiveness of the lab sessions was evaluated based on an objective comparison between the exam results of the old and the new system. Significant improvements were observed in both courses: the average grades improved by 0.54 and 0.9 grades in the two courses respectively (on a scale from 1 to 5). In one of the courses, the participation rate at the oral exam also increased. Encouraged by these achievements there are plans for introducing computer proof assistants in other courses.

The high-level precise modelling of programming languages is not only an essential part of proving program correctness but also helps Computer Science students gain proficiency in programming languages in general and improve their abstraction skills. Hopefully, the experiences shared in this paper can provide useful advice on how to apply proof assistants in theoretical courses, making them more accessible to students.

Acknowledgement

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Ambrus Kaposi was also supported by the New National Excellence Program of the Ministry for Innovation and Technology, Project no. ÚNKP-19-4-ELTE-874, and by the Bolyai Fellowship of the Hungarian Academy of Sciences, Project no. BO/00659/19/3.The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Bibliography

1. The Agda programming language (from 2007)
   https://wiki.portal.chalmers.se/agda/pmwiki.php
2. Tobias Nipkow, Gerwin Klein: *Concrete Semantics with Isabelle/HOL* (from 2014)
   http://concrete-semantics.org/
3. Coq In The Classroom (from 2017)
   https://github.com/coq/coq/wiki/CoqInTheClassroom
4. GitHub repository of the Coq proof assistant (from 2007)
   https://github.com/coq/coq
5. The Coq proof assistant (from 1989)
   https://coq.inria.fr/
6. The Haskell programming language (from 1990)
   https://www.haskell.org/
7. The Isabelle proof assistant (from 1986)
   https://isabelle.in.tum.de/
8. Donkó István: orsi-formalization. Student formalization of material from the Distributed Systems course at ELTE. (2019)
   https://github.com/Isti115/orsi-formalization

9. Benjamin C. Pierce et al.: *Programming Language Foundations*. Software Foundations series, volume 2. Electronic book. (2018) http://www.cis.upenn.edu/~bcpierce/sf

10. Philip Wadler: *Programming Language Foundations in Agda*. Conference material, 21st Brazilian Symposium (SBMF 2018). Salvador, Brazil, November 26–30, 2018.

11. O. Johansson: *Programming language semantics*. Course materials. Umeåi University, Umeå, Sweden (1993-1998) http://oldwww.cs.umu.se/local/kurser/TDBC05/

12. Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press, USA.

13. Ambrus Kaposi. 2019. Nyelvek típusrendszere. Lecture notes https://bitbucket.org/akaposi/tipusrendszerek/src/master/

14. Per Martin-Löf. 1982. *Constructive Mathematics and Computer Programming*. In: Editor(s): L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, Klaus-Peter Podewski. Studies in Logic and the Foundations of Mathematics, Elsevier, Volume 104, 1982, Pages 153-175.

15. Philip Wadler. 2015. *Propositions as types*. *Commun. ACM* 58, 12 (November 2015), 75–84. DOI: https://doi.org/10.1145/2699407

16. Xavier Leroy & Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy and Jérôme Vouillon. 2020. *The OCaml system release 4.10: Documentation and user's manual.*

Authors

BERECZKY Péter

Eötvös Loránd University, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary, e-mail: berpeti@inf.elte.hu

DONKÓ István

Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, e-mail: isti115@inf.elte.hu

HORPÁCSI Dániel

Eötvös Loránd University, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary, e-mail: daniel-h@elte.hu

KAPOSI Ambrus

Eötvös Loránd University, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary, e-mail: akaposi@inf.elte.hu

About this document

Published in:

CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE

Volume 2, Number 1. 2020.

ISSN: 2676-9425 (online)

DOI:

10.36427/CEJNTREP.2.1.470

NÉMETH Dávid János

Eötvös Loránd University, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary, e-mail: ndj@inf.elte.hu

License

Copyright © BERECZKY Péter, DONKÓ István, HORPÁCSI Dániel, KAPOSI Ambrus, NÉMETH Dávid János*, 2020.*

Licensee CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE, Hungary. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) license.

http://creativecommons.org/licenses/by/4.0/