Towards a Mechanized Theory of Computation for Education

Tiago Cogumbreiro\textsuperscript{1} and Yannick Forster\textsuperscript{2}

\textsuperscript{1} University of Massachusetts Boston, Boston, Massachusetts, U.S.A.
\textsuperscript{2} Inria, France

tiago.cogumbreiro@umb.edu

Abstract

In this talk we present a mechanization effort of theory of computation in the context of undergraduate education, with a focus on decidability and computability. We introduce a Coq library used to teach 3 sessions of a course on Formal Languages and Automata, at the University of Massachusetts Boston. Our project includes full proofs of results from a textbook, such as the undecidability of the halting problem and Rice’s theorem. To this end, we present a simple and expressive calculus that allows us to capture the essence of informal proofs of classic theorems in a mechanized setting. We discuss the assumptions of our formalism and discuss our progress in showing the consistency of our theory.

Introduction. Formal languages and automata (FLA) is in the basis of the curriculum of undergraduate computer science \cite{10}. We report on an open source project written in Coq \cite{3} to mechanize results of classical theory of computation. The first author used this software for 3 semesters to teach decidability, computability, and regular languages at the University of Massachusetts Boston. Proof assistants play a central role in our lectures for three reasons. Firstly, a proof assistant offers an interactive mechanism to allow students to step through a proof autonomously, allowing students to independently browse every detail of a proof at their own pace. Secondly, a proof assistant turns a logic assignment into a programming assignment, which can be more approachable to computer science students. Thirdly, having proof scripts that can be machine checked, lets instructors automatically grade homework assignments. Other works that use proof assistants to aid education include \cite{1, 9, 12}.

Mechanization goals. We formalize Sipser’s Introduction to the theory of computation \cite{16} in Coq. Our design goal is to keep our formalism as close to the textbook as possible, which includes having mechanized proofs that mirror the textbook proofs. Another important design goal is that proofs should only include basic Coq capabilities. The proofs need to be comprehensible to an undergraduate student with rudimentary knowledge of Coq (case analysis, induction, polymorphism, and logical connectives). Further, we include alternative proofs of some theorems when there’s a pedagogical benefit, e.g., the proof is simpler, or the intuition is easier to explain. Our approach contrasts many published works on mechanized computability theory \cite{13, 17, 8, 2, 15, 4, 11}.

Decidability results. Our mechanization includes the main results of \cite[Chapters 4 and 5]{16}, on decidability and reducibility. One of our contributions is formalizing Sipser’s “high-level descriptions,” which is essentially pseudo-code to describe a Turing machine. For instance, consider Theorem 4.11 of \cite[Chapter 4]{16}, where $\text{TM}$ denotes a Turing machine, ranged over by meta-variable $M$, inputs are ranged over by $i$. A language, ranged over by $A$ is a set of inputs. Language $A$ is decidable if there exists a Turing machine $M$ that decides $A$, i.e., $M$ accepts $i$ if, and only if, $i \in A$; and $M$ rejects $i$ if, and only if, $i \notin A$. Additionally, $\langle \cdot \rangle$ denotes a reasonable encoding of one or more objects into a string, e.g., $\langle M, i \rangle$ encodes a Turing machine $M$ and an input $i$ into an input, and $\langle M \rangle$ encodes a Turing machine $M$ into an input.
Theorem 4.11 ([16, pp. 207]). \( A_{TM} = \{ (M,i) \mid M \text{ is a TM and } M \text{ accepts } i \} \) is undecidable.

The proof of Theorem 4.11 includes the following high-level description of a Turing machine \( D \), parameterized by a Turing machine \( H \) which decides \( A_{TM} \): “The following is the description of \( D \): (1) Run \( H \) on input \( (M,\langle M \rangle) \). (2) Output the opposite of what \( H \) outputs. That is, if \( H \) accepts, reject; and, if \( H \) rejects, accept.”

We formalize such high-level description as:

**Definition D** \( (H:\text{input} \to \text{prog}) \): \( \text{input} \to \text{prog} := \) 

\[
\begin{align*}
\text{fun } (i:\text{input}) & \Rightarrow (\text{* On input } i = \langle M \rangle \text{ *)} \\
\text{mlet } b & \leftarrow H \langle \text{decode_mach } \ i, i \rangle \ \text{in} \ (* \text{ Step 1 } *) \\
\text{if } b & \text{ then Ret false else Ret true} \hspace{1cm} (* \text{ Step 2 } *)
\end{align*}
\]

where input \( i = \langle M \rangle \) and \( \text{decode_mach } \ i = M \). We formalize high-level descriptions next. We first give the syntax of high-level descriptions \( p \).

\[
p ::= \text{mlet } x = p \ \text{in} \ p \mid \text{call } M \ i \mid \text{return } b \hspace{1cm} \text{where } b \in \{ \top, \bot \}
\]

Next, we introduce a big-step operational semantics in terms of high-level descriptions:

\[
\begin{array}{cccc}
\text{return } b & \downarrow & \text{M accepts } i & \downarrow \top \\
\text{call } M \ i & \downarrow & \text{M rejects } i & \downarrow \bot \\
\text{p \ downarrow b} & \Rightarrow & \text{p'}[x := b] \downarrow b' \\
\text{mlet } x = p \ \text{in} \ p' & \downarrow & \bot & \downarrow b'
\end{array}
\]

We then define that a Turing machine \( M \) computes a Turing function \( f \) of type \( \text{input} \to \text{prog} \) if for any input \( i \) we have \( \text{call } M \ i \downarrow b \) if and only if \( f(i) \downarrow b \).

**Assumptions.** Our theory is parameterized by an input type, a type of Turing machines, and the semantics of Turing machines. We assume that their execution is deterministic and that for any machine \( M \) and an input \( i \) we can obtain \( M \) accepts \( i \), or \( M \) rejects \( i \), or neither. Centrally, we assume that for any Turing function \( f \) there exists a Turing machine \( M \) computing \( f \). This assumption is consistent since we work in Coq, where every definable function is computable.

**Results.** To mechanize the proof of Theorem 4.11 we assume a machine \( M' \) deciding \( A_{TM} \), i.e., computing a Turing function \( H \). Now using the assumption that every Turing function is computable on \( \mathcal{D}(H) \) yields \( M \) such that \( \text{call } M \ i \downarrow \text{true} \leftrightarrow \text{call } M \ i \downarrow \text{false} \), a contradiction.

Our further main results include: \( A \) is decidable if, and only if \( A \) is recognizable and co-recognizable (i.e., its complement is recognizable) (Theorem 4.22); the complement of \( A_{TM} \) is not recognizable (Corollary 4.23); \( \text{HALT}_{TM} = \{ (M,i) \mid M \text{ accepts or rejects } i \} \) is undecidable (Theorem 5.1); \( F_{TM} = \{ (M) \mid L(M) = \emptyset \} \) is undecidable (Theorem 5.2), where \( L(M) = \{ i \mid M \text{ accepts } i \} \); \( \text{EQ}_{TM} = \{ (M_1,M_2) \mid L(M_1) = L(M_2) \} \) is neither recognizable nor co-recognizable (Theorem 5.30); \( \text{EQ}_{TM} \) is undecidable (Theorem 5.4); Rice’s Theorem (Problem 5.28). Our proofs of these results are all constructive. Our project contains results on languages and regular languages, e.g., the pumping lemma for regular language inspired by the proof in [14].

**Future work.** To show the consistency of our axioms, we are working on instantiating our theory with a mechanized formalism of computability from the Coq library of undecidability proofs [7] equivalent to Turing machines [6]. Consistency of the central assumptions then follows from the consistency of the axiom \( CT \) in type theory [5]. We are also investigating multiple grading approaches in classes that use proof assistants, e.g., multiple-choice questions, automatic questions about students submissions.
References


