Teaching Induction with Functional Programming and A Proof Assistant

Peter-Michael Osera   Steve Zdancewic

University of Pennsylvania

{posera, stevez}@cis.upenn.edu

Abstract

Mathematical induction is a difficult subject for beginning students of computer science to fully grasp. In this short paper, we propose using functional programming and proof assistants as an aide in teaching mathematical induction in a traditional discrete mathematics course. To demonstrate this approach, we created a proof-of-concept web-based tutorial on induction. In this tutorial, students write small functional programs and prove simple properties about them using inductive reasoning. The functional programming language is deliberately designed to be minimalistic so that it can be picked up quickly, especially if the student is already familiar with a functional programming language, and not be a distraction to the ultimate goal of learning induction. Furthermore, the tutorial features an online IDE for entering programs and proofs to minimize the barrier to entry for students and instructors.

1. Introduction

Mathematical induction and formal proof are notoriously difficult subjects for undergraduates in computer science to fully grasp. Many students walk out of their discrete mathematics, formal methods, and algorithms courses without really understanding what constitutes a proof by induction, let alone when it is necessary. Worst yet, even if students come away with a sense of how to carry out inductive proofs, they don't understand how their knowledge of formal proof and inductive reasoning can inform their day-to-day programming tasks.

We believe that the programming languages community possesses two unique contributions to solve this problem:

Functional Programming Inductive datatypes, recursive functions, and (structural) induction share a strong connection with each other. Exploiting this connection can solidify understanding among all three topics.

Proof Assistants Mechanizing proofs unveils the hidden structure and nuances of propositions much like how programming does the same for algorithms. Proof assistants can help students understand the proper structure of formal proofs, in particular inductive proofs.

Several universities have integrated functional programming early in their curriculum, some as early as the introductory programming sequence. With the addition of functional programming to the 2013 Computer Science Curricula revision [7], many more universities will be integrating functional programming into their programs. While functional programming is an inherently valuable subject for the computer science student on its own, we must explore how knowledge of functional programming can enhance students' learning throughout their computer science education. Induction is certainly one such topic that synergizes well with functional programming.

The benefits of proof assistants such as Coq [12], Agda [6], Twelf [9], and Isabelle [13] are well known to the programming languages community. Proof assistants help us verify the metatheoretic claims that we make and frequently reveal subtle, game-breaking flaws in our logic. However, we are also well-aware of the shortcomings of proof assistants: they are complicated to use and burdensome for anything but the smallest of developments.

This has not stopped educators from integrating proof assistants into the classroom, for example programming language foundations [10] and formal logic [5]. However, these efforts have been "heavyweight" in that sense that they have introduced the proof assistant in its full glory to the student. As a result, learning how to use the proof assistant itself becomes as much of the course as the rest of the content. Because induction is only one part of a traditional discrete mathematics course, it is not prudent to introduce a full-fledged proof assistant just to teach induction more efficiently.

1.1 Contributions

To teach the basics of formal proof by induction, we only need a small subset of the proof assistant's functionality. Therefore, in this short paper we propose a new learning module to teach induction. With this module, we write simple functional programs and then prove properties about those programs with the aide of a proof assistant. We believe this module has a number of benefits:

1. The inductive datatypes of a functional programming language and inductive reasoning are a natural fit for each other. Coupling the two shows students why induction is important not just as a theoretical device, but as a reasoning tool for practical programming.
2. Using a proof assistant makes the process of proof concrete and explicit to the student similarly to how programming makes concrete the process of algorithmic thinking.
3. Proof assistants themselves have an addictive, game-like quality to them that helps the student focus on examples and exercises.
4. Indirectly, using a proof assistant raises the important question of informal versus formal proof. Except in this context, students

SPLASH-E '13 October 28, 2013, Indianapolis, IN
Copyright © 2013 ACM . . . $15.00

have a very concrete notion of formal proof, thanks to the proof assistant, to base this conversation.

Traditional proof assistants raise the barrier to entry substantially, so to lower this barrier and demonstrate our module, we have implemented MINIFN, a prototype programming system consisting of a small, core functional language and proof assistant. The system contains just enough features to explain inductive proofs in the context of functional programs. With this minimalistic approach, we hope to make adoption by instructors easier. To further ease adoption, we have implemented MINIFN as a web service that requires no installation on the part of the student. Finally, we have implemented a proof-of-concept online tutorial on induction that uses the MINIFN system as an embedded IDE.

Section 2 outlines our proposed learning module in more detail. Section 3 gives a brief over the MINIFN system. Section 4 describes our prototype online tutorial, and we discuss our future directions in Section 5.

2. The Module

In this section, we describe our induction module in more detail. In particular, we define our target audience, goals, and then give an outline of the module content itself.

2.1 Audience

Our approach to teaching induction targets the beginning computer science undergraduate learning induction in the context of a discrete mathematics course. In light of the 2013 curriculum revision [7], we ideally would like the student to already be exposed to functional programming. However, the functional programming component of this module is small enough that a student only exposed to a traditional introductory programming course can pick it up without much effort.

Alternatively, this module can be used by students who are just learning functional programming. Visiting proof by induction in this context reinforces the strong mathematical foundation of functional programming as well as showcases the ease at which we can reason about programs that possess this sort of foundation.

2.2 Goals

In combining functional programming and proof assistants, it is easy to digress into programming language theory minutiae that, while interesting, are ultimately irrelevant to the goal of teaching induction. To prevent this from happening, we have clearly defined goals with this approach to ensure that we stay on track in spite of these new technologies. Students should:

1. Learn the *mechanics* (form and structure) of a proof by structural induction,
2. Learn *when* induction is a necessary proof technique and useful reasoning tool,
3. Learn the distinction between structural induction and mathematical induction (of which the latter is a special case of the former),
4. Learn/reinforce some of the fundamentals of functional programming: inductive datatypes and recursion,
5. Understand the connection between functional program design with inductive datatypes and recursive functions and inductive reasoning,
6. Understand the spectrum that exists between formal, machine-checked proof and informal reasoning, and
7. Understand how a strong sense of the formalities of inductive proof influences day-to-day informal reasoning about programs.

In particular, there are a number of things we do not touch upon. Our module, while using functional programming, is not a replacement for a complete treatment of functional programming. In particular, in the course of the module, we have no need of first-class and higher-order functions. While this is a good opportunity to discuss the deep connection between proof and programs (i.e., the Howard-Curry Isomorphism), we avoid this discussion entirely in the interest of focusing our discussion. Finally, we also do not leverage our proof assistant to teach additional aspects of formal logic other than what is necessary to establish basic proofs of induction.

2.3 Module Outline

With these goals in mind, our induction module proceeds as follows.

Functional Programming We first introduce/review inductive datatypes, pattern matching, and structurally recursive functions over those datatypes. In particular, we use simple datatypes — booleans and natural numbers — as well as lists over these datatypes. At this stage, we expect students to have a good idea of the *mechanics* of recursive functions. That is, they understand that a chain of recursive function calls "bottom out" at a defined base case. This intuition is important because we use it to justify why the "inductive assumption" of an inductive proof is also well-founded.

Simple Proofs We then discuss proving simple propositions about programs using these datatypes. To prove these properties, students use simplification and case analysis over inductive datatype constructors. At this stage, we use the MINIFN proof assistant, but we stress the student first produce proofs on paper and then check their proofs with MINIFN. The formal MINIFN proof quickly becomes the template that students use to write their proofs.

For example, one exercise we present is double negation elimination for booleans, that is $\neg(\neg b) = b$ for any boolean b. The paper proof we demand of the student is relatively verbose:

- Consider a particular b. This b must be either true or false.
- If $b = \mathsf{true}$ then we have $\neg(\neg\mathsf{true}) = \mathsf{true}$. If $b = \mathsf{false}$ then we have $\neg(\neg\mathsf{false}) = \mathsf{false}$ which completes the proof.

However, it mimics what the MINIFN proof assistant requires of them and thus enforces this connection between the logic they need to go through when proving these propositions and what the proof assistant provides.

Induction We show that case analysis quickly becomes insufficient when reasoning about inductive datatypes, and the "inductive assumption" is what we need to to make our proofs go through. We describe induction as case analysis but with the addition of this assumption. We then justify the correctness of this assumption by comparison to the structurally recursive function calls we have been writing up to this point. We then go through a variety of examples using induction over natural numbers, motivating the distinction between traditional mathematical induction and structural induction, and lists. Two such examples are proving that addition over the natural numbers is monotonic and list reversal preserves list length.

Formal vs. Informal Proof In order to help students lift their experience of writing inductive proofs in MINIFN to other situations, we close by discussing the difference between formal and informal

$$\begin{array}{rcl}
\mathit{Decls} & ::= & \mathsf{type}\ D = C_1\ T_{11} \ldots T_{1n} \mid \ldots \mid C_k\ T_{j1} \ldots T_{jm} \\
 & \mid & \mathsf{let}\ f(x_1 : T_1) \ldots (x_n : T_n) : T = e \\
T & ::= & T \mid T_1 \to T_2 \\
e & ::= & \mathsf{fun}\ (x_1 : T_1) \ldots (x_n : T_n) : T = e \mid e_1\ e_2 \\
 & \mid & \mathsf{match}\ e\ \mathsf{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n \\
\mathit{prfcmd} & ::= & \mathsf{introduce} \mid \mathsf{simplify} \mid \mathsf{destruct}\ e \\
 & \mid & \mathsf{rewritef}\ h \mid \mathsf{rewriteb}\ h \mid \mathsf{induction}\ e
\end{array}$$

Figure 1. The MINIFN programming language.

proof. We offer the paper proofs they have written so far as a way to structure their formal proofs for future classes. We also recommend that when reasoning about inductive structures in their own programs that they structure their informal reasoning in the style of a formal proof (but, of course, not go into that level of detail).

3. The Minifn System

We want to avoid overwhelming students with the complexities of a full-fledged programming language and proof assistant or otherwise distract them away from the goals we have outlined thus far. To do this, we utilize a prototype programming system called MINIFN which features a core statically-typed functional programming language as well as a basic scripting language for writing proofs about MINIFN programs. MINIFN itself lowers to Coq [12], so the system is backed by a full-fledged proof assistant.

With MINIFN, we want to give programmers just enough power so that they can write simple, yet interesting recursive datatypes and functions over those datatypes. Figure 1 gives the syntax of MINIFN. To accomplish this, we only need datatype declarations, (structurally) recursive functions, and pattern matching. Other features such as let-binding, type inference, and polymorphism are not necessary and thus not included. Our hope is that this core language is simple enough that it is immediately familiar to anyone that knows a pre-existing functional language or easily picked up by anyone that has taken an introductory programming course.

MINIFN's proof language employs a subset of the Coq vernacular as shown in Figure 1. This subset is just enough to be able to prove the simple inductive properties that we would like. For example, the monotonicity of addition property is proven as follows:

```
Lemma plusMonotonic : forall (n:nat) (m:nat),
  gte (plus n m) n = true
Proof
  introduce
  induction n

  simplify
  rewrtitef gte0
  reflexivity

  simplify
  rewritef IHn
  reflexivity
Qed
```

The `plusMonotonic` property claims that:

$$\forall n, m : n + m \geq n$$

The first two lines of the proof state this property in MINIFN language. The remaining lines are a *proof script* outlining the steps of the proof. If we were to write the proof on paper, it might look like this:

- To prove this fact, we choose particular natural numbers (`nats`) n and m and proceed by induction on n. A natural number is either 0 or $1 + n'$ for some (smaller) natural number n', so it is sufficient to prove the claim for each case.
- When $n = 0$ then $0 + m \geq 0$ is true because $0 + m = 0$ for any m.
- When $n = 1 + n'$ then $1 + n' + m \geq 1 + n'$. Our induction hypothesis states that $n' + m \geq n'$. After simplification, it is sufficient to prove that $n' + m \geq n'$ which is exactly what our induction hypothesis says.

Proof scripts are composed of a series of commands which are drawn from a subset of the Coq tactics language. What is nice is that the proof script above follows the paper proof exactly. If you are already familiar with Coq, then you can likely see how the formal and paper proofs coincide. For those less familiar, it is worthwhile to illustrate this correspondence:

- In the first block, the `introduce` commands moves the variables `n` and `m` under the `forall` quantifier and makes them assumptions analogous to our "choosing" particular natural numbers n and m. The `induction n` command performs induction on `n` breaking up the proof into two parts: when `n` is zero and when `n` is the successor of some `n'`.
- In the second block, we prove the case where `n` is zero. `simplify` simplifies our goal by performing as much computation as possible. The fact that `plus 0 m` is `m` for any `m` is recorded in an auxiliary lemma `gte0` that we use to rewrite the goal with the `rewritef` command (where the `-f` implies a *forward* rewrite). This leaves us with a syntactic identity left to prove (*ie.*, `true = true`), so we use the `reflexivity` command to discharge this goal.
- In the third block, we prove the case where `n` is the successor of some `n'`. The case proceeds similarly to the previous case except that we use the inductive hypothesis (named `IHn`) to rewrite the goal.

What is nice about this subset is that it closely mimics the paper proof we want students to write. Thus, students are not "learning the proof assistant" but instead using MINIFN to structure and check their formal reasoning. In particular, MINIFN allows the student to check that the induction hypothesis they think they possess is correct, a common mistake that beginning students make.

4. Online Tutorial

To put our module into practice, we have developed a proof-of-concept online induction tutorial. You can view the current iteration of the tutorial at this address:

`https://fling.seas.upenn.edu/~posera/induction`

The tutorial is intended for beginning computer science students who are being exposed to induction and formal reasoning for the first time. It mirrors the outline of the module given in Section 2:

1. We first start with an overview of the MINIFN programming language. Students who have not yet been exposed to functional programming follow an introduction to the core functional programming concepts needed for our module. Students who already have exposure to functional programming can instead choose to read a short overview of MINIFN's functional programming language.
2. Next, we introduce basic proofs by way of an introduction to MINIFN's proof language. We incrementally build up more interesting propositions from trivial ones to show the purpose of each MINIFN proof command and also impart the structure of proof onto the student.

3. We then proceed to the heart of the tutorial: induction. In the tutorial, we motivate induction by giving a simple example where case analysis, `destruct`, fails because it leads to an endless cycle of `destructs`.
4. We finally close the online tutorial with a discussion of informal and formal proof.

Keeping in line with our learning objectives, a major theme of this tutorial is the connection between paper proofs and mechanized proofs that the students write in MINIFN. In this tutorial we stress that students develop proofs on paper as they normally would and then use the MINIFN proof system as a checker. By exposing the student to formal logic checked by a proof assistant, we hope that students gain insight into the mechanics of induction and confidence to carry out these proofs correctly on paper. And by using the MINIFN programming language to motivate these proofs, we hope that students come to realize the deeper connection between inductive datatypes and inductive reasoning.

This prototype tutorial uses an online-based implementation of MINIFN. Figure 2 gives an example of this widget as it is embedded within the tutorial. The MINIFN language is type-checked in-browser, and the resulting AST is sent to the server where it is translated to Coq, evaluated, and then sent back to the browser in a processed form. These widgets are interspersed throughout the tutorial to allow students to quickly try out examples and exercises. We hope that this online IDE reduces the barrier to adoption for both students and educators.

5. Conclusion

Our induction module is still a work-in-progress. We close with related work and our future directions for this module.

5.1 Related Work

Our work is inspired by two other educational efforts here at the University of Pennsylvania. The first is our efforts to introduce functional programming into our CS2 course [14] which is an exemplar in the 2013 curriculum revision [7]. This course features the OCaml programming language in its first half to give students a firm mathematical foundation to their programming endeavors as well as level the playing field among the diverse students we find in our introductory sequence. The second is our graduate-level programming languages course, Software Foundations [10], which features the Coq proof assistant. In particular, the course features a rapid introduction to functional programming in Gallina, Coq's specification language, and proving basic properties over those specifications with Coq's tactics language. Our module can be seen as a refinement of this process, tailor-made for the purpose of learning induction.

Other universities have also adopted functional programming early in the curriculum. Most notably, Carnegie Mellon University's Principles of Functional Programming course [1], features a module on proving properties of simple functional programs. In this case, the properties are proven on paper only, without the aid of a proof assistant.

Finally, proof assistants themselves have been used as the primary learning vehicle in a variety of classes. The aforementioned Software Foundations course at the University of Pennsylvania is one such example. A more closely related example to our work is the ProofWeb online proof assistant [5]. ProofWeb acts as an online front-end for a variety of proof assistants and has been used in several classes ranging from formal logic to type theory. ProofWeb exposes the full power of the proof assistant to the user which makes it too heavyweight for our more modest purposes.

Notably, the University of Oklahoma and Northeastern both have used ACL2 in the context of software engineering and introductory programming courses to expose students to the power of combining programming with theorem proving [8]. In particular, the ACL2 Dracula [3] and the ACL2 Sedan (ACL2s) [2] projects demonstrate the potential of employing theorem proving to link theory with programming. Like MINIFN, Dracula and ACL2s both provide interfaces to the students that tightly interweave the act of programming and proof. However, where these efforts differ from MINIFN is that MINIFN requires the student build the proof manually whereas ACL2 automatically searches for the proof. This difference reflects the different philosophies of the two projects. The efforts around ACL2 Dracula and ACL2s are designed to expose students to software verification with practical theorem proving tools whereas MINIFN strives for the more modest goal of helping students understand the mechanics of inductive proofs and bridging formal and informal reasoning.

5.2 Future Work

Our induction module and the MINIFN system are both in the early proof-of-concept stage. We are currently refining the flow of the module, the examples we present, and layout and presentation of the tutorial.

5.2.1 Improvement

As mentioned previously, MINIFN is backed by the Coq proof assistant. The initial version of MINIFN is heavily influenced by this decision. In particular, the MINIFN proof commands are taken from the Coq tactics language. While this has made our proof-of-concept easier to implement, these commands may not be ideal as a surface language for the student. For example, we likely want to force the student to explicitly name the cases they expect when performing a `destruct` or `induction`, *e.g.*, `induction: n = O, n = S n'`, something that Coq does not natively provide. This way, the proof script that they write better reflects the structure of the proof. Other syntactic improvements are certainly possible to provide the student with a better experience.

Another area of improvement is extending the tutorial to cover more advanced topics regarding induction proofs. For example, one of the most difficult concepts for students is knowing when to strengthen their induction hypothesis. We believe that machine-checked proofs will help demystify this process for the student. However, the addition of more complicated proofs will likely require that we extend MINIFN with extra machinery or commands which will need to balanced against keeping the system tractable for students to learn.

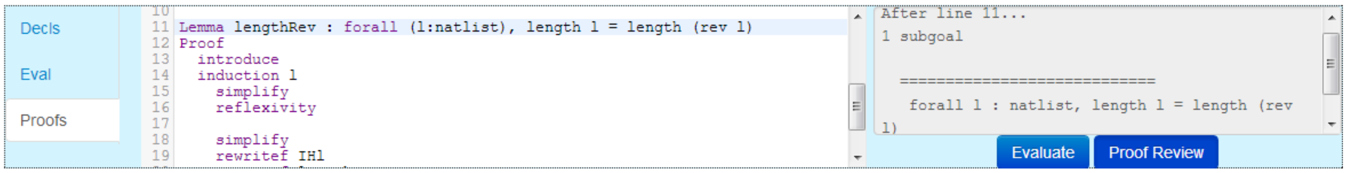
One final area of improvement, and one of our motivations for pursuing this line of work, is employing the techniques of program synthesis to automatically generate practice induction problems and solutions as well as give feedback to the student [4]. We believe that our approach gives us additional leverage in this domain because our choice of using Coq as the back-end means that our proofs and programs have a "programmatic" representation in the Calculus of Inductive Constructions that can be amendable to existing synthesis techniques with some adaption, *e.g.*, sketching [11].

5.2.2 Deployment

We plan on releasing our work along two vectors:

1. Internally as an augmentation to the induction portion of our discrete mathematics class (CIS 160) in the spring and
2. Externally as a publicly available tutorial usable by anyone over the Internet.

The University of Pennsylvania introductory programming sequence (CIS 120, in particular) uses the OCaml programming language, so the syntax of MINIFN closely resembles OCaml to ease



```
11 Lemma lengthRev : forall (l:natlist), length l = length (rev l)
12 Proof
13   introduce
14   induction l
15     simplify
16     reflexivity
17
18     simplify
19     rewritef IHl
```

```
After line 11...
1 subgoal

  ============================
   forall l : natlist, length l = length (rev
l)
```

Figure 2. The MINIFN online widget.

the transition into our module. This fall we are planning to pilot the tutorial with a select group of students and TAs from CIS 160. After integrating their feedback into our tutorial, we plan to widen our scope to the whole class. In this setting, we would like to evaluate the efficacy of our module with respect to our current, more traditional approach to teaching induction. We hypothesize that we will be able to leverage our student's existing knowledge of OCaml to make learning induction more concrete and directly applicable to what the student already knows: programming.

Furthermore, one of the major appeals of this work is its potential to scale well to situations outside of the UPenn classroom. We plan on continuing to refine our web offering of the module so that interested educators can use it either as a supplement or even potential replacement to their current methods for teaching induction. We also hope that by offering our module on the web that interested students and computer science practitioners that are interested in strengthening their induction skills find it a useful resource.

Acknowledgments

We thank Val Tannen for assisting us in piloting this module for CIS 160, the testers over the Internet that provided feedback and bug reports on the initial version of this module, and the reviewers of SPLASH-E for their invaluable feedback on this project.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-138996 "Expeditions in Computer Augmented Program Engineering (ExCAPE): Harnessing Synthesis for Software Design." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Cs 15-150: Principles of functional programming. URL `http://www.cs.cmu.edu/~15150/`.

[2] H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon. The acl2 sedan theorem proving system. In *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, TACAS'11/ETAPS'11, pages 291–295, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19834-2. URL `http://dl.acm.org/citation.cfm?id=1987389.1987424`.

[3] C. Eastlund and M. Felleisen. Automatic verification for interactive graphical programs. In *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, ACL2 '09, pages 33–41, New York, NY, USA, 2009. ACM. ISBN 9781-60558-742-4. doi: 10.1145/1637837.1637843. URL `http://doi.acm.org/10.1145/1637837.1637843`.

[4] S. Gulwani. Example based learning in computer-aided stem education. Technical Report MSR-TR-2013-50, Microsoft Research, 2013.

[5] C. Kaliszyk, F. V. Raamsdonk, F. Wiedijk, M. Hendriks, and R. D. Vrijer. Deduction using the proofweb system, 1999.

[6] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[7] T. J. T. F. on Computing Curricula. *Computer Science Curricula 2013 — Ironman Draft (Version 1.0)*. Association for Computing Machinery and IEEE-Computer Society, February 2013.

[8] R. Page, C. Eastlund, and M. Felleisen. Functional programming and theorem proving for undergraduates: a progress report. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*, FDPE '08, pages 21–30, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-068-5. doi: 10.1145/1411260.1411264. URL `http://doi.acm.org/10.1145/1411260.1411264`.

[9] F. Pfenning and C. Schuermann. *Twelf User's Guide*, 2002.

[10] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2012.

[11] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html`.

[12] T. C. D. Team. *The Coq Proof Assistant — Reference Manual*, 2013.

[13] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2013.

[14] S. Zdancewic and S. Weirich. Programming languages and techniques: Lecture notes for CIS 120, April 2013. URL `http://www.cis.upenn.edu/~cis120/current/notes/120notes.pdf`.