ACL2 for Freshmen: First Experiences

Carl Eastlund cce@ccs.neu.edu
Dale Vaillancourt dalev@ccs.neu.edu
Matthias Felleisen matthias@ccs.neu.edu

College of Computer Science
Northeastern University
Boston, MA 02115

Abstract

Northeastern University's College of Computer Science uses an applicative subset of Scheme in its introductory programming course with a heavy emphasis on design. Students then proceed to a second-semester programming course using Java and a course on symbolic logic. During the 2007 spring semester, we experimented with an ACL2-based course as a potential replacement for the freshman course on symbolic logic. This paper reports on the specifics of the experiment (context, syllabus, students), its mixed outcomes, and our conclusions for future revisions. Based on our preliminary experiences, the College has decided to adopt our experimental course as the standard course.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

DrScheme, ACL2, formal methods, pedagogy

Keywords

DrScheme, ACL2, TeachScheme!

1 Logic in the Computing Curriculum

Introducing undergraduate students to logic poses a major challenge to computer science departments. Although logic is to computing what analysis is to physics, the designers of undergraduate curricula still assign logic a minor role in the overall educational program. Many curricula relegate logic to a single required course or a module in a course on discrete mathematics. Worse, few of those courses connect logic to the students' active programming experience, which usually means that students fail to appreciate the importance of logic for their future work.

Northeastern University's College of Computer Science recognizes the importance of logic and requires freshman students to enroll in Symbolic Logic, a course offered by the Philosophy department. This service course is targeted at computer science students and taught by a faculty member of the Philosophy department who has an adjunct appointment in the College. Even so, the course is a classical introduction to propositional and predicate logic, with little connection to computing. As Felleisen has observed over five years, undergraduates simply cannot connect the content of Symbolic Logic with any software-related activity.

In response to this observation, Felleisen proposed to develop and test an alternative course that combines ACL2 programming with elements of Symbolic Logic. The conjecture is that since students learn to design functional programs in the first semester course in Scheme, it would be easy to write the same functions in Applicative Common Lisp (ACL), to state conjectures about them in the Computational Logic, and to prove them automatically or with minimal interaction. Students thus motivated could then be introduced to the logical underpinnings that enable proofs about programs, and they would learn to appreciate that logical guidelines—such as those taught in the freshman programming course—lead to correct and well-designed programs.

This paper reports on our experience with a small field test of this conjecture. At the end of Fall '06, we asked for six volunteers from the freshman course and ended up with 28. Of those, we picked two students with stellar records, two with weak As, and two with B-level final grades. These students signed up for directed studies but we actually conducted a small regular course. All students were concurrently enrolled in Symbolic Logic. The results of our course are highly encouraging but also point to weaknesses in the approach. Still, the College has found the results convincing and has decided to take on the Symbolic Logic course and to use our approach as a starting point.

Section 2 presents the pedagogic and technological context of the experiment. Sections 3 and 4 make up the heart of the paper. They present the material we taught and the evaluation of the course. The last two sections discuss related and future work.

2 Background and Context

Northeastern's freshman year in computer science has the following structure:

Form of data / Steps	atomic	enumeration	records	unions	inductive	…
1. Data definition						
2. Purpose & contract						
3. Functional examples						
4. Template (inventory)						
5. Function definition						
6. Examples to tests						

For each form of data, fill in the appropriate questions:
Q1a: Does the data definition include disjoint subsets?
  A: If so, formulate a conditional with as many clauses.
    Q1b: What are the distinguishing conditions for each clause?
Q2: Is any of the data structured?
  A: If so, formulate selector expressions.
Q3: Does the data definition refer to itself?
  A: If so, introduce recursion in the matching clause and selector expression.

Figure 1. The two dimensions of the design recipe.

Programming and Computing	Mathematical Context
CS U 211 *Fundamentals I* functional program design, algebraic models of computation	CS U 200 *Discrete Structures* discrete mathematics, including sets, functions, relations, inductive proofs
CS U 213 *Fundamentals II* class-based program design, object-oriented computation	PHL U 215 *Symbolic Logic* introduction to propositional and predicate logic, models

The courses in the left column introduce students to rigorous program design principles in two language paradigms: functional programming (using an applicative dialect of Scheme) and object-oriented programming (using a dialect of Java, dubbed ProfessorJ [9]).

The purpose of the courses in the right column is to introduce students to those mathematical topics that are useful to practicing computer scientists. The first course mostly covers mathematics for algorithmic topics, e.g. combinatorics; the second is a course on logic that contains little computing material. Students without the necessary calculus background (some 30%) are registered in a remedial calculus course and start the right-column sequence one semester behind the regular students.

For the introduction of an ACL2-based logic course in place of PHL U 215, the existence of CS U 211 is a necessity. In this course, students learn to design programs according to a "design recipe" that systematically matches the inductive structure of the input. In addition, the design recipe introduces a process with intermediate products of high diagnostic value. Roughly speaking, the design recipe is somewhat analogous to "The Method" promoted by the ACL2 literature [10, 11].

A second prerequisite for teaching novices is appropriate technology, especially a programming environment that supports the gradual introduction of linguistic constructs with a special focus on error messages. Since a proper analysis of this issue is beyond the paper, we refer the reader to literature on DrScheme [8]. Based on our DrScheme experience, we developed Dracula, a pedagogic programming environment for ACL2; the environment proved invaluable for introducing freshman students to ACL programming.

In order to provide the complete background to our experiment, the first subsection provides a summary of the pedagogy in CS U 211, based on Felleisen et al's text [5]. The second subsection introduces the essential elements of Dracula [14].

2.1 How to Design Programs

The CS U 211 course is based on Felleisen et al's TeachScheme! project and the resulting text (*How to Design Programs*) [6, 7]. Based on observations of numerous high school and college courses, Felleisen et al concluded that programming novices need to see more than the syntax of a programming language and a few programming examples to play with. What novices should immediately see is the systematic design of programs.

To this end, the textbook introduces the idea of a *design recipe*. The design recipe is a six-step process for problem solving that evolves with the complexity of data representations over the course of a semester. Figure 1 depicts the process and its evolution as two "dimensions" of the design recipe. Along the vertical axis, the table lists the process-oriented part of the recipe. Along the horizontal axis, it enumerates the forms of data to which the process steps are applied.

The design recipe starts with a thorough analysis of the problem. The goal of this first step is to describe the collections of data that the program may consume and produce. For the second step, the novice programmer is to formulate the tasks in his/her own words; this includes a "contract," i.e., a statement about the input and output data/conditions. The third step is to create small functional examples that illustrate what the program computes from given data. Following this, the fourth step brings in the second dimension of the design recipe. The goal is for the programmer to construct a "template" for the program's organization from the data definition of the input(s). Put differently, they write down the inventory of givens because it is those pieces of data that the program can use to produce its answer. Then, and only then, students code. The very fact that coding is the fifth step of the design recipe—even for small programs—shows students that the creation of good programs requires a solid preparation, just like cooking a good meal. Last but not least, students must turn the functional examples into a test suite. Doing so tends to reveal typos, misunderstandings and other simple mistakes. Of course, passing tests merely ensures that the program performs as expected on small examples. Still, students may only run the program—apply to large and/or a lot of real data—after they have completed this sixth step.

Clearly, the fourth step, template construction, is the critical one. It demands a thorough understanding of data definitions and how they relate to the organization of programs. The course staff supports this step with the question-and-answer "game" in figure 1. Indeed, they use these questions all the time and never respond to students' questions with explicit answers. Eventually students realize that they can ask those questions themselves and the question-and-answer game becomes second nature.

Given the central role of data definitions, it is natural that the course is organized along the complexity of data definitions instead of the syntax of the programming language. Specifically, *How to Design Programs* uses naïve set theory as the basis of this development. It starts with atomic data (numbers, booleans, characters, symbols) where the data definition says nothing about the organization of the function. Instead, knowledge about the application domain dictates what the function definition looks like. As the course progresses, the data definitions become more highly structured—such as an inductive definition of a tree—and the structure of the data definition determines in most cases almost all the program structure. For details, we refer the reader to Felleisen et al's article on the design rationale for their book and course [6].

The alert reader has recognized by now that the goal of the template is to train students to recognize inductive definitions of data and to translate them into structural recursions. Let's illustrate this idea briefly. Say the student has determined that the function's input is a list of dates:

```
(define-struct date (year month day))
;; A Date is (make-date Number Number Number)


;; A LOD (list of dates) is one of:
;; – empty
;; – (cons Date LOD)


;; MyFunction : LOD -> ... [intentionally omitted detail] ...
;; MyFunction returns ... [intentionally omitted detail] ...
(define (MyFunction an-lod) ... )
```

The first question of the design recipe (figure 1, **Q1**) demands the introduction of a **cond** expression into the body with two clauses:

```
(define (MyFunction an-lod)
  (cond
    ((empty? an-lod) ... )
    ((cons? an-lod) ... )))
```

Answering the second question (**Q2**) reminds the student that *empty* (a.k.a. *null*) is atomic and *cons* is structural, with *first* and *rest* (a.k.a. *car* and *cdr*) used as selectors:

```
(define (MyFunction an-lod)
  (cond
    ((empty? an-lod) ... )
    ((cons? an-lod) ... (first an-lod) ... (rest an-lod) ... )))
```

Finally, students introduce a recursive call around (*rest an-lod*) in response to the third question (**Q3**):

```
(define (MyFunction an-lod)
  (cond
    ((empty? an-lod) ... )
    ((cons? an-lod) ... (first an-lod) ...
                    ... (MyFunction (rest an-lod)) ... )))
```

In short, the question-and-answer game guides students to use the program structure that LISPers find natural and use intuitively. The difference is that students can take away design rules that also work in an object-oriented setting (CS U 213), something they are likely to encounter in the rest of the curriculum and their internships.

For purely motivational reasons, the course has heavily relied on interactive graphical games for the last five editions. The design recipe works well for this setting and students respond with enthusiasm. In support of this scenario, DrScheme supports images as first-class data and an algebra of images for manipulating them.

Still in the first-order world of data, students also encounter the concept of *generative recursion* in the world of *How to Design Programs*. In contrast to structurally recursive functions, generatively recursive functions do not traverse the given data according to its structure. Instead, they generate a new piece of data (of the same kind) and recur on it. For example, while insertion sort proceeds structurally, quick sort proceeds in a generative fashion. At that point, the text also extends the design recipe process with a seventh step: a termination analysis. After all, generatively recursive functions can diverge for some inputs, unlike structurally recursive functions, which always terminate. Thus, students have also encountered an essential element of ACL2's admission process and are prepared to understand that for some functions this analysis is straightforward and for others it is complex.

Accumulator-style programming is the last ingredient from *How to Design Programs* that is relevant here. Once students have seen generative recursion, an instructor can demonstrate how, for example, an applicative algorithm for searching in a graph may diverge for cyclic graphs. This flaw naturally motivates accumulator-style programming. Like generatively recursive functions, accumulator-style programs perform recursion with newly generated values, motivating strengthened induction hypotheses for proofs.

Students come away from CS U 211 with several important ideas. First, programming requires a systematic design process. Second, the central step of the design process demands a solid understanding of shapes of data definitions. They also get the idea that logic is relevant for both axes of design. Finally, they learn to apply the first two ideas in the context of applicative Scheme. We conjecture that freshmen with this background can succeed in a course using ACL2.

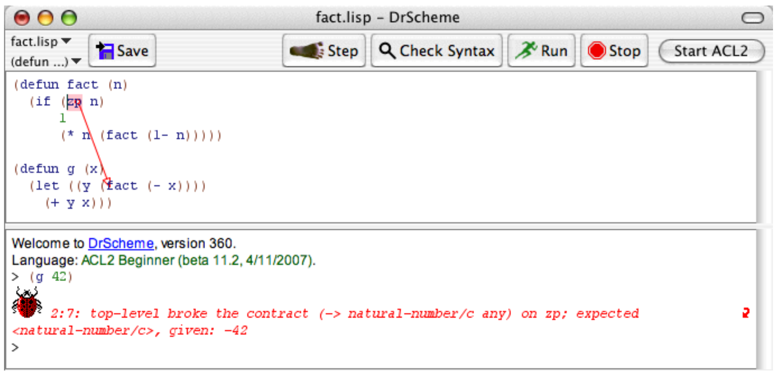
2.2 Dracula

Dracula [14] is an embedding of a subset of Applicative Common Lisp into DrScheme. The embedding simulates ACL using PLT Scheme's macro and syntax system, and therefore inherits DrScheme's tools, especially the interactions window (read-eval-print loop) and the Check Syntax tool. Downloads, instructions, and more information on Dracula can be found at `http://www.ccs.neu.edu/home/cce/acl2/`.

Based on our experience with the TeachScheme! project, we decided to make Dracula's primitives safe and to enforce safety at all times.[1] We consider safety essential for novice programmers, so that they discover "type" problems as quickly as possible. Furthermore, freshman students are used to Scheme and may have encountered Java in high school, both of which are safe programming languages.

Figure 2 displays a snapshot of Dracula. For novices, the IDE consists of three parts: a console with five buttons for controlling the

[1] In ACL2 terminology, guard-checking is on.

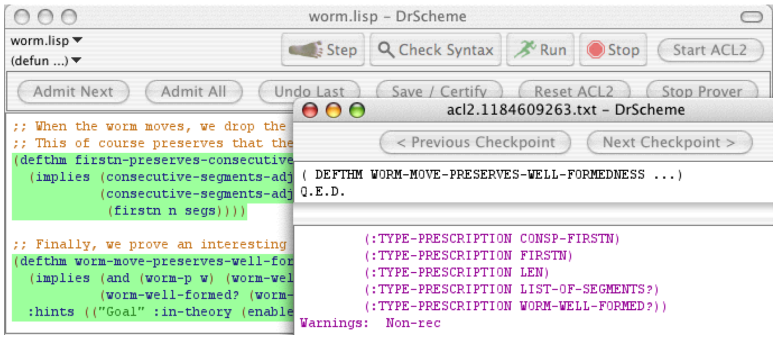


```
(defun fact (n)
  (if (zp n)
      1
      (* n (fact (1- n)))))

(defun g (x)
  (let ((y (fact (- x))))
    (+ y x)))
```

```
Welcome to DrScheme, version 360.
Language: ACL2 Beginner (beta 11.2, 4/11/2007).
> (g 42)
2:7: top-level broke the contract (-> natural-number/c any) on zp; expected
<natural-number/c>, given: -42
>
```

Figure 2. Safety check violation in DrACuLa



```
;; When the worm moves, we drop the
;; This of course preserves that the
(defthm firstn-preserves-consecutive
  (implies (consecutive-segments-adj
           (consecutive-segments-adj
            (firstn n segs))))

;; Finally, we prove an interesting
(defthm worm-move-preserves-well-for
  (implies (and (worm-p w) (worm-wel
           (worm-well-formed? (worm-
  :hints (("Goal" :in-theory (enable
```

```
( DEFTHM WORM-MOVE-PRESERVES-WELL-FORMEDNESS ...)
Q.E.D.
```

```
        (:TYPE-PRESCRIPTION CONSP-FIRSTN)
        (:TYPE-PRESCRIPTION FIRSTN)
        (:TYPE-PRESCRIPTION LEN)
        (:TYPE-PRESCRIPTION LIST-OF-SEGMENTS?)
        (:TYPE-PRESCRIPTION WORM-WELL-FORMED?))
Warnings:  Non-rec
```

Figure 3. Interacting with ACL2 via DrACuLa

IDE, plus an optional Save button; a definitions window; and an interactions window.

The definitions window is a Scheme-oriented program editor, which in this case contains function definitions for *fact* (factorial) and *g*. Once the student clicks the RUN button, the definitions are evaluated[2] and the focus shifts to the bottom pane, the interactions window.

The interactions window is roughly like a Lisp read-eval-print loop, though Dracula reinitializes the state of the loop every time a student clicks RUN.[3]

In the screen shot, the student is trying to evaluate (g 42). Since g then calls *fact* with -42, the safety check for *zp*, whose domain proper is the set of natural numbers, raises an exception. Dracula highlights the use of *zp* that causes the error and, with a failure arrow, also shows the pending call to *fact* and where its result would be used (through y and **let**).

To prove theorems, Dracula invokes the standard ACL2 implementation, rather than simulating its logic. A student can launch the theorem prover with a click on Start ACL2 (top right). As the screen shot in figure 3 shows, doing so brings up a "report window" (on the right) and opens an additional command pane in Dracula, with buttons for sending definitions and theorems from the definitions window to the theorem prover.

[2]Dracula can optionally prove the function total by invoking the ACL2 theorem prover during this process.

[3]This so-called *transparent* read-eval-print loop is inherited from DrScheme. Our experience with freshman courses suggests that creating a clean slate is critical [8].

The additional console in the DrScheme window supports six new functions. The first button sends the next unadmitted definition or theorem to ACL2. If the prover accepts it, Dracula highlights it in green; otherwise it turns red. The second button sends the entire definitions window to ACL2, one definition at a time. The third button undoes the last admit. The fourth button attempts to certify the current file as an ACL2 book. The fifth button resets the theorem prover to its initial state. And the last one terminates the ACL2 process. At the time of the course, this was the only way to interrupt a proof.

The ACL2 report window has two panes. The top pane shows the proof tree summary of ACL2's proof attempts, adapted from similar functionality in the ACL2 Emacs mode. The bottom pane shows the complete output of the ACL2 session. The buttons at the top of the window, Previous Checkpoint and Next Checkpoint, allow users to jump directly to key points of the latest proof attempt in the bottom pane.

3 Course Description

Vaillancourt and Eastlund taught the experimental course CS U 294 on formal logic and automated theorem proving in the spring semester of 2007. The course consisted of three parts: lectures, homeworks, and a final project. There were no exams in the course. Here we outline our plan for the course, how we executed it, and how the students performed.

3.1 Conjecture and Motivation

The working conjecture of our project is that proving theorems about working programs motivates students to perceive logic as the

1. Introduction to ACL2 programming
2. Propositional logic
3. Model theory and interpretations
4. Structural induction principles
5. Structural induction proofs
6. ACL2 tips: `defstructure`
7. Formal proof language
8. Inference rules
9. Proving induction principles
10. Induction for generative recursion
11. Induction for accumulators
12. Model theory for first-order logic

Figure 4. Lecture topics.

"calculus of computer science." To be precise, we expected that the selected students had understood functional program design, both along the data and the process axes. In addition, the introductory programming course repeatedly makes claims about programs that are easily formulated and proven in a logic such as ACL2. Thus, students should be ready for the transition from programming to proving theorems about programs.

Based on this conjecture, we intended to start the logic course with simple exercises on programming familiar functions in ACL2 instead of Scheme, formulating claims as theorems, and proving them automatically in ACL2. From there, the course could then proceed to theorems that do not go through automatically, which would demand a close look at the nature of claims, proofs, and logic in general.

Students were expected to become proficient in evaluating logical propositions, relating them to computer programs, proving them by hand, and establishing simple proofs in ACL2. In a broader sense, we expected that students would take away two lessons from this course:

1. Reasoning about programs requires logic in the same way that reasoning about physical events requires calculus.
2. Attempting to prove theorems is just another step in detecting and preventing flaws in programs and their structure.

While we still believe that such a logic course could be designed and taught, our first experiences suggest that such an effort demands an even tighter integration of programming (as in CS U 211) and logic (as in CS U 294) than we imagined.

3.2 Lecture Material

The course syllabus combined the theory of formal logic with practical applications to program verification. The complete progression of topics is shown in figure 4. We started with a primer on programming in ACL2. We followed with propositional logic, including conjunction, disjunction, implication, atomic propositions, and truth tables. We introduced model theory via interpretations, which map atomic propositions to truth values.

The course continued with a treatment of structural induction. First we showed students how to create a structural induction principle from a recursive data definition, and walked through several examples. The following lecture used these rules for some examples of inductive proofs.

Figures 5 and 6 show an example data definition and structural induction principle for lists of numbers, as presented in lecture. We taught students to formulate the induction principle by case analysis on the clauses of the data definition. This process mimics the template step of the design recipe from CS U 211. Figure 7 shows the program `sum` which adds the elements of a list of numbers; figure 8 demonstrates a proof, also taken from the lecture material, of a simple property of `sum`. The proof follows the form of the induction principle: one clause for `nil` and one for $(\texttt{cons n l})$; the second clause proves an implication based on an inductive hypothesis (I.H.).

We spent the next lecture on tactics for using ACL2. We started by presenting `defstructure`, a mechanism from one of the ACL2 books for reasoning about structured data as presented in CS U 211. The lecture continued with strategies for recovering from failed proof attempts in ACL2, which we repeated with examples throughout the semester. Students were encouraged to work out proofs by hand ahead of time. We taught them to compare ACL2's output to their solution on paper, and to guide ACL2 with lemmas where it deviated from the expected strategy. Dracula's Previous and Next Checkpoint buttons helped students jump to the key points in the output of failed ACL2 proof attempts.

The course transitioned to focus on proof theory. First, we discussed a formal language of proofs and how they can be formally checked. Then we presented the notation of inference rules. We gave examples of their use, including translating structural induction principles from implications to inference rules and redoing prior proof examples with them.

Next, we examined the foundations of induction principles. We

$$\text{LoN} = \texttt{nil} \mid (\texttt{cons}\ \text{Number LoN})$$

Figure 5. Data definition for a List-of-Numbers (LoN).

if $P(\texttt{nil})$ and $\forall \texttt{l} \in \text{LoN}.\ P(\texttt{l}) \Rightarrow \forall \texttt{n} \in \text{Number}.\ P((\texttt{cons n l}))$
then $\forall\ \texttt{l} \in \text{LoN}.\ P(\texttt{l})$

Figure 6. Structural induction principle for a LoN.

```
(defun sum (l)
  (cond ((endp l) 0)
        ((consp l) (+ (car l) (sum (cdr l))))))
```

Figure 7. Simple program operating on LoN.

$\forall\ \texttt{xs}, \texttt{ys} \in \text{LoN}.$
$(\texttt{sum}\ (\texttt{append xs ys})) = (+\ (\texttt{sum xs})\ (\texttt{sum ys}))$
Proof by induction on `xs`.

Case $\texttt{xs} = \texttt{nil}$:

$(\texttt{sum}\ (\texttt{append nil ys}))$	
$= (\texttt{sum ys})$	definition of `append`
$= (+\ 0\ (\texttt{sum ys}))$	additive identity
$= (+\ (\texttt{sum nil})\ (\texttt{sum ys}))$	definition of `sum`

Case $\texttt{xs} = (\texttt{cons n l})$:
I.H.: $(\texttt{sum}\ (\texttt{append l ys})) = (+\ (\texttt{sum l})\ (\texttt{sum ys}))$

$(\texttt{sum}\ (\texttt{append}\ (\texttt{cons n l})\ \texttt{ys}))$	
$= (\texttt{sum}\ (\texttt{cons n}\ (\texttt{append l ys})))$	definition of `append`
$= (+\ \texttt{n}\ (\texttt{sum}\ (\texttt{append l ys})))$	definition of `sum`
$= (+\ \texttt{n}\ (+\ (\texttt{sum l})\ (\texttt{sum ys})))$	I.H.
$= (+\ (+\ \texttt{n}\ (\texttt{sum l}))\ (\texttt{sum ys}))$	associativity of `+`
$= (+\ (\texttt{sum}\ (\texttt{cons n l}))\ (\texttt{sum ys}))$	definition of `sum`

Figure 8. Inductive proof over LoN.

proved the validity of structural induction in terms of set theory and infinite unions, then generalized our notion of induction by showing how any closure condition for a set can be an induction principle. Structural induction allowed us to reason about structurally recursive programs, which recur on a component of their input. Expanding our set of induction principles allowed us to reason about generative recursion.

To exemplify generative recursion, we presented an implementation of quicksort. We formulated a new induction principle for lists based on their length, rather than their construction, and used it to prove equivalence between quicksort and insertion sort, except for a few lemmas left as exercises. This introduced students to strong induction.

Another variation on recursion, programs which use accumulators, was our next example. An accumulator is an argument to a function which records the result so far and is constructed, rather than destructured, by the computation. We presented the implementation of summing a list that records the total so far as an accumulator, rather than the naive structurally recursive implementation, which builds up a call stack of additions. We used this to motivate the tactic of strengthening induction principles: the naive induction based on the structure of the list fails to take the changing accumulator into account.

We wrapped up our material on program verification before assigning the final project and concluded the semester with two lectures on model theory for first-order logic. We covered universal and existential quantifiers, logical structures defining a set of objects and predicates, and how to establish the truth of a proposition in a given structure.

3.3 Lecture Practices

The course met twice weekly during the semester, for 60 minutes on Tuesday mornings and 45 minutes on Friday afternoons. Usually, we used Tuesdays to present new material and Fridays as lab sessions. In lab sessions, students were allowed to work on their homework in pairs and ask questions as needed.

Lectures were informal and interactive; because of the small class size, we prompted each student to participate during every lecture. After presenting each new topic, we worked through a few examples as time permitted. At each step we asked students to work out the next part of a proof or program, and discussed why each choice was correct or incorrect.

Lab sessions were less structured. Students brought laptop computers and used the class as supplemental homework time. We answered questions they raised and presented new ACL2 techniques to the class as appropriate.

3.4 Homework Assignments

We assigned students eight homeworks over the course of the semester. The progression of topics is shown in figure 9. The assigned problems were a mix of ACL2 programming, proofs by hand, and proofs with ACL2. We had students work and submit as pairs, changing the partnerships a few times during the semester. After the pairs handed in their assignments, we discussed the solutions in class. In some cases, we had students present their program or proof to the class and had the rest of the class critique the solution.

1. ACL2: Basic programming
2. ACL2: Propositional validity checker
3. Hand-written: Model theory and structural induction
4. ACL2: Structural induction on binary search trees
5. ACL2: Proof checker and more structural induction
6. Hand-written: Why structural induction fails for quicksort
7. Hand-written: Lemmas for proof about quicksort
8. ACL2: Accumulators and generative recursion

Figure 9. Homework topics.

We designed the first assignment to give students familiarity with ACL2. We asked students to represent two simple data structures, implement a few structurally recursive functions over them, and formulate a test suite for their program. We gave instructions for running the program in both Dracula and ACL2.

The second assignment built on students' ACL2 programming skills and the model theory lecture material. Students were instructed to represent a language of propositions in ACL2 and implement predicates such as validity and satisfiability over them. The students had to derive their own data definition for interpretations (a mapping from atomic propositions to truth values) to solve the problems.

The third assignment consisted of several proofs to be done by hand. The first two theorems were simple equivalences in model theory. The remaining problems required students to formulate structural induction principles for recursive data types and prove theorems about programs written either in class or on the previous homework.

The fourth assignment applied the material on structural induction to ACL2 proofs. Students had to prove that a binary search tree insertion procedure produces a new binary search tree containing one extra occurrence of the given element and preserves the tree's ordering invariant. This was the first ACL2 proof required of students; the assignment provided several hints, and students were expected to show up for help in office hours as needed.

The fifth assignment consisted of three programs and two proof obligations. The first two parts presented straightforward structurally recursive programs and correctness properties for students to prove. The third part gave a simple language for proof trees in propositional logic and asked students to implement a proof checker for it. There was no ACL2 proof required for the proof checker.

One part of this assignment was problematic. We required students to prove that insertion sort produces a permutation of its input. ACL2 admitted the staff solution in moments without helper lemmas, but apparently correct student solutions failed after long proof attempts. Our normal recourses of adding intermediate lemmas or working out the proof by hand turned out to be intractable, as the full proof was too complicated to write in the allotted time. ACL2 had found a shortcut we could not duplicate, and we were unable to help students make progress short of handing them a complete solution.

The sixth assignment asked students to write an explanation of why structural induction will not work for proofs about a quicksort implementation. Students had to attempt the proof and discover where the inductive hypothesis failed to be helpful. We intended the exercise to prepare students for the material on alternate induction principles.

In the seventh assignment, we asked students to finish a proof equating the results of quicksort to those of insertion sort; we had reduced the proof to six lemmas during lecture. This was a pencil and paper assignment. After it was due, we asked each student to present one of the proofs to the rest of the class. Students critiqued each others' methods and worked out corrected proofs for each lemma as needed.

The eighth and final assignment required several proofs in ACL2 using nontrivial induction techniques. Three required students to write a recursive program with an accumulator and prove a correctness property. The fourth asked students to prove the equivalence of quicksort and insertion sort, previously worked out by hand, using ACL2.

We planned to judge from students' performance and feedback on homeworks how well they were learning ACL2, what difficulties they encountered, and how much the design recipe benefitted their proof attempts. We expected that students would continue to apply the design recipe as they learned it in CS U 211. To this end, we stressed a policy of testing before theorem proving. We supplied students with a testing teachpack[4] that gathered and summarized the results of test cases.

We also expected students to use the ACL2 techniques we discussed in class, such as writing proofs by hand and comparing the proof structure to ACL2's output. Students were instructed to seek extra help from the instructors as needed so that they could continue to make progress and we could see what obstacles they encountered.

Unfortunately, the students did not perform as per our expectations. They normally, but not always, used the testing teachpack as we instructed and stated contracts and purposes for their functions; they rarely followed the more structured steps of the design recipe. Students were reluctant to seek extra help on their own, yet also frequently skipped attempts to prove theorems by hand or interpret the output of ACL2 from a failed proof attempt. Many students instead adopted their own trial-and-error methods of developing proofs.

3.5 Final Project

The final project for the course was an open-ended programming and theorem proving assignment. We provided students with the implementation of a tetris-like video game in Dracula; initially, a single block dropping down a blank screen and falling off the bottom. We implemented the graphical interface to the game with an animation teachpack, based on the one students had used in the previous semester. The interactive portions would execute in Dracula but were ignored by ACL2.

The supplied file included a correctness property, which the program violated: that the block would remain within the visible area of the screen. We asked students to fix the program to satisfy the property in class on the day we handed out the project, to get them familiar with the code. Like the regular homeworks, students worked on the project in pairs.

We gave students two weeks to add features to the game and prove interesting properties of the final program. We requested tetris-like features—users should be able to move blocks as they fall, blocks should stack up at the bottom of the screen, blocks should not overlap, etc.—but left the final decisions up to students. We checked on each pair's progress after a week and encouraged students to seek extra help as needed. Figure 10 shows an example snapshot of a tetris-like game as played from a student's submission.

[4]Teachpacks are libraries of code for students to use. They operate like books when run by ACL2, and may have extra interactive effects when run by Dracula.

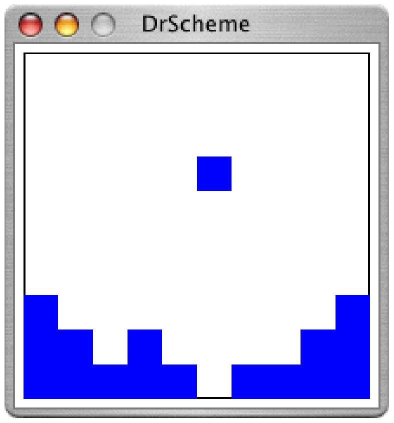


Figure 10. A sample tetris-like game.

After the due date, we collected the programs from students and had them demonstrate the final product in class. Students demonstrated the features they incorporated into the game, the properties they had attempted to prove, and the status of their proof attempts.

Performance on the final project mirrored that of the homeworks. While the groups successfully implemented the program features we suggested and a few others, their adherence to the design recipe was incomplete, which hindered their proof efforts. Each group successfully proved one or two nontrivial safety conditions of their program. The homeworks also each contained one failed proof attempt. In each case, students were unable to interpret the output of ACL2, had not attempted the proof by hand, and had not contacted the course staff for extra help.

4 Summative Evaluation

The course concluded at the end of the Spring 2007 semester. After we assigned final grades to the students, we formulated and conducted exit interviews with all students individually.

4.1 Student Perspective

At various points during the semester, students commented that the class using ACL2 was an improvement over Symbolic Logic and more rewarding than most other courses. They also had some complaints and suggestions for the course, which they provided in detail during the exit interviews.

In general, students found the course faster paced and more challenging than their other first-year coursework. They said that our course progressed rapidly to more advanced topics of formal logic than the Symbolic Logic course. Students were enthusiastic about the course, often voluntarily staying beyond the scheduled lecture time to continue discussing the material.

All of the students found the interface to ACL2 too verbose; the first few checkpoints of a failed proof attempt often started in a manner they found sensible, but quickly transitioned to checkpoints with

Topic	Notation
BNF Grammar	Tree ::= Atom | $(\texttt{cons}\ \text{Tree Tree})$
Structural Induction Principle	$\dfrac{\begin{array}{c}\forall a : \text{Atom}.(\texttt{P}\ a) \\ \forall x,y : \text{Tree}.(\texttt{P}\ x) \wedge (\texttt{P}\ y) \Rightarrow (\texttt{P}\ (\texttt{cons}\ x\ y))\end{array}}{\forall x : \text{Tree}.(\texttt{P}\ x)}$
Validity	$\models A \vee \neg A;\ \not\models A \vee B$
Modelling	$\models_\alpha A \wedge \neg B;\ \alpha = \{A = \top, B = \bot\}$
Entailment	$A, A \Rightarrow B, B \Rightarrow C \models A \wedge B \wedge C$
Set Theory	$D = \bigcup_{n:\mathbb{N}} L^n(\emptyset)$

Figure 11. Logical notation introduced in CS U 294.

large subgoals or unexpected strategies they were not able to diagnose. Some students got discouraged early in the course and did not continue to read the output of ACL2 to discover the problems in their proofs.

Five of the six students expressed a desire for more in-class training on proof development and use of ACL2. They felt unprepared to make some of the key decisions needed for nontrivial proofs, such as how to choose lemmas, when to abandon a theorem, or when to attempt a proof by hand.

Two students also stated they felt overwhelmed by the presentation of formal logic. The theory material introduced a lot of new notation that students had not seen before and was not present in ACL2. Figure 11 contains several examples. Students had several suggestions to improve this, such as providing a course syllabus and either textbooks or complete lecture notes, or presenting more of the logical theory in terms of ACL2 and programming notation.

4.2 Instructor Perspective

The results of the course were mixed. We were not able to adequately evaluate the impact of CS U 211 on students' ability to use ACL2 effectively. We did, however, teach a group of freshman of various academic standings to use ACL2 productively and independently in the course of a semester, concluding by proving multiple nontrivial properties of a meaningful interactive program.

Part of our initial thesis for this course was to discover how effective the design principles taught in CS U 211 can be in the instruction of formal reasoning about programs. We experienced some success in this regard: our treatment of induction principles and structural induction followed the pattern of templates and recursion from the prior course, and students picked it up readily. Unfortunately, the homeworks and projects submitted for the course clearly lacked many design recipe elements. As a result we have not satisfactorily explored our thesis; freshmen require strong incentive and instruction to adhere to structured programming practices.

We also intended to build on students' prior programming experience via Dracula, providing a familiar interface through DrScheme to the new language of ACL2. The tool was clearly helpful to the class; students were able to develop proofs and programs without the learning curve of a new development environment, and had a rich set of tools such as the testing and animation teachpacks. However, Dracula was missing some key functionality. Some controls for the graphical interface did not function properly, including control over navigation of ACL2 output. Dracula's Help Desk lacked complete documentation; it provided ACL2's documentation, but did not list what subset it implemented nor provide tutorials for common tasks.

The course's greatest challenge was the difficulty of recovering from a failed proof attempt. Part of the problem was our curriculum; we presented a series of individual tactics for completing proofs, rather than a single step-by-step method. Some of the presented tactics, such as specific ACL2 books to include or proof hints to use, came too late, and students got stuck on homeworks without knowing why. Students were also reluctant to admit failure and ask an instructor for help, despite encouragement and the experimental nature of the course.

Much as we would like it to, the difficulty of generating proofs in ACL2 does not always scale down with the complexity of the problem. Regardless of the degree of training, ACL2 proof attempts are fragile, and homework exercises require extremely careful design to help students avoid common obstacles to successful proofs. As we discovered with the fifth homework assignment, some proofs may be trivial by one approach and intractible by another. Teaching ACL2 requires a large canon of simple, provable program properties without this drawback.

Perhaps the most surprising success of this course was the students' performance at in-class presentations of proofs. Students learned quickly how the instructors expected a proof to be presented; by the end of the first set of presentations, the class was able to construct a proof collaboratively even when the presenting student did not have a correct proof to start from.

Students took our course alongside PH U 215, Symbolic Logic. While we presented enough formal logic for students to reason about programs, the Symbolic Logic course gave students a fuller treatment of pure logic than our course. We took advantage of students' dual enrollment to focus on ACL2-related topics; a full course based on our curriculum could cover more formal logic.

Over the course of a semester, we taught formal logic and ACL2 theorem proving to six students spread across the top half of their class, with two or more instructors present for all lectures and lab sessions. We demonstrated methods which were overall very successful at teaching the theory of proofs, and which had some difficulty with the practice of using ACL2. For inclusion into a required freshman course, this curriculum must be adapted for a larger class size with less instructor attention. While our experience does not yet represent a repeatable way to teach ACL2 to a large class of students, our results suggest that freshmen can learn and apply automated reasoning about programs, and give us clear directions for future improvement.

5 Related Work

Over the past ten years, NSF has funded three efforts on introducing students to logic in programming-related ways. They are the ACL2-based project run by J Moore and P. Manolios, the Beseme Project at the University of Oklahoma [13], and TeachLogic [1], a distributed project.

The main pedagogic result from Moore and Manolios's project is "The ACL2 Sedan", a customization of the development environment Eclipse for learning ACL2, developed at Georgia Tech by Dillinger, Manolios, and Vroon [4]. The ACL2 Sedan gives new degrees of control over the interface to ACL2; for instance, users have the option of a specialized termination analysis [12], or customizing the set of proof search features used by default. We are interested in incorporating these and other features of ACL2s into future versions of Dracula.

The Beseme Project, developed by Rex Page, presents a curriculum for undergraduate discrete mathematics courses that applies formal reasoning tools to Haskell programs. The curriculum has been taught at the University of Oklahoma. Page confirmed with statistical correlations an improved performance in subsequent, required computer science courses.

TeachLogic, a joint project by Barland (Rice), Felleisen (Northeastern), Kolaitis (UCSC), and Vardi (Rice), consists of a collection of teaching modules designed to incorporate elements of logic into a variety of computer science topics. The project espouses "logic across the curriculum", by which formal reasoning can be spread throughout undergraduate education and applied directly to real software. The modules are published and available from the web site; the TeachLogic team has not attempted to gather data about the pedagogic long-term effects of the modules.

6 Future Work

Our experience with this course suggests a number of future software projects as well as projects for developers of courses and curricula on formal reasoning in computer science.

A revision of the course must insist that students stick to the functional design recipe. Thus the course can serve as a reinforcement of the recipe, and students are likely to make progress on the proof-oriented material more quickly.

The syllabus must evolve to include a structured theorem-proving method analogous to the design recipe. Specifically, if ACL2 is to be retained, it requires exercises for interpreting ACL2's diagnosis, for selecting intermediate lemmas, and for building proofs incrementally. The course will also benefit from a large canon of carefully designed exercises that demonstrate proof principles while avoiding ACL2 subtleties along the way, such as perplexing failure output or the need to use mysterious, instructor-supplied "hints".

Concerning Dracula, we intend to focus on two major complaints. On one hand, there is clearly a need for improved documentation of common tasks and expanded explanations of some basic ACL2 functions and forms. The ACL2 tutorial, included with Dracula, is a good starting point for professionals but assumes a level of technical sophistication beyond most freshmen. On the other hand, Dracula (and ACL2) needs a better tool for navigating ACL2's explanations. The existing checkpoint controls in Dracula do not function reliably, and the information found at each checkpoint is still too complex for a novice programmer to decipher. We will therefore attempt to develop a tool for navigating and restructuring ACL2's output so that users can get piecemeal explanations and summaries as needed.

We have also considered some entirely new tools that would help students develop programs and proofs using Dracula. A stepper for Dracula in the style of DrScheme's student languages [3] would help students see their programs in action. A test generation facility such as QuickCheck for Haskell [2] could help students find counterexamples to faulty conjectures more quickly than a full-fledged proof attempt. Better support in the programming language of Dracula for incremental and modular development of both proofs and programs would help students break down large problems into small tasks, and ease the progress from small proofs to large projects.

Acknowledgements. Thanks to Rex Page for inspiring this course, to Kathi Fisler for inspiring the senior author to pursue this idea in 1998, and to Felix Klock for his assistance in the classroom. We received comments on early drafts from Peter Dillinger, Pete Manolios, Riccardo Pucella, Olin Shivers, Sam Tobin-Hochstadt, and anonymous referees.

7 References

[1] Barland, I., M. Felleisen, P. Kolaitis and M. Vardi. The teachlogic project. See `http://www.teachlogic.org/`.

[2] Claessen, K. and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[3] Clements, J., M. Flatt and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 22–37, 2001.

[4] Dillinger, P. C., P. Manolios, D. Vroon and J. S. Moore. ACL2s: "The ACL2 Sedan". *Electron. Notes Theor. Comput. Sci.*, 174(2):3–18, 2007.

[5] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[6] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. In Hanus, M., S. Krishnamurthi and S. Thompson, editors, *Functional and Declarative Programming in Education*, 2002.

[7] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14:55–77, 2004.

[8] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[9] Gray, K. E. and M. Flatt. ProfessorJ: A gradual intro to Java through language levels. In *OOPSLA Educators' Symposium*, 2003.

[10] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[11] Kaufmann, M., P. Manolios and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[12] Manolios, P. and D. Vroon. Termination analysis with calling context graphs. In Ball, T. and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006.

[13] Page, R. L. Software is discrete mathematics. In *ACM SIGPLAN International Conference on Functional Programming*, pages 79–86, 2003. Also: `http://www.cs.ou.edu/~beseme/`.

[14] Vaillancourt, D., R. Page and M. Felleisen. ACL2 in DrScheme. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 107–116, New York, NY, USA, 2006. ACM Press.