Design and development of Proof Pad, A Pedagogic Development Environment for the ACL2 Theorem Prover

Caleb Eggensperger

University of Oklahoma
calebegg@gmail.com

Abstract. Integrated Development Environments (IDEs) are an increasingly important part of how students and professionals write code. The availability of free, high quality, industry-grade IDEs for popular imperative languages has led to a high expectation of quality in programming tool sets that is often difficult for less mainstream languages to meet. ACL2 (A Computational Logic for Applicative Common Lisp) is a Lisp dialect and theorem prover that suffers from this problem. ACL2 is a powerful system with potential for classroom use in a way that encourages and develops the ability to write code which can be tested and reasoned about in a rigorous way.
This paper discusses Proof Pad, a new IDE for ACL2. Proof Pad is not the only attempt to provide ACL2 IDEs catering to students and beginning programmers. The ACL2 Sedan and DrACuLa systems arose from similar motivations. Proof Pad builds on the work of those systems and on pedagogic IDEs such as DrScheme and BlueJ. It also takes into account the unique workflow of the ACL2 theorem proving system.
As Proof Pad is a teaching tool first and foremost, the design and in-progress implementation of Proof Pad incorporated student feedback from the outset, and that process continued through all stages of development. Feedback took the form of direct observation of users interacting with the IDE as well as questionnaires completed by users of Proof Pad and other ACL2 IDEs. The result is a streamlined interface and fast, responsive system that supports using ACL2 as a classroom tool.

1 Introduction and Prior Work

1.1 ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) is a Lisp dialect and theorem prover in the Boyer-Moore family of theorem provers. Functions and theorems in ACL2 are stated using an applicative (purely functional) subset of Common Lisp, where variables are read-only, and functions must be written using a recursive rather than an iterative style. Because of this, user-stated theorems can be more easily proven by the automated theorem prover from the collection of theorems and lemmas available in the system (both built-in and user-defined). Using this mechanism, it's possible to "steer" ACL2 to prove complex theorems with minimal interaction [1]. ACL2 has predominantly been used to model and verify the hardware and sometimes software of critical computer systems.

Recently, Carl Eastlund [2] and Rex Page [3][4][5] have used ACL2 as a pedagogic software development tool. Page has been using ACL2 and DrACuLa (an ACL2 IDE) in the University of Oklahoma's software engineering course since 2003. ACL2 was brought into the course to aid the students with good design and defect control in their projects. The approach has been successful, with students able to create projects of significant complexity with a solid proof-based underpinning [3].

For this two-semester course, students were asked to approach a series of small individual and team projects, leading up to a final, semester-long team project. Projects begin with the creation of short functions and the formalization of simple theorems, in order to develop students' ability to state and prove properties of their code. The course culminates with a project of 2,000 to 3,000 lines, along with ACL2 properties for individual components, documentation, unit tests, and integration tests [3].

1.2 Existing ACL2 development environments

The development of Proof Pad was informed by existing tools for working with ACL2. The most common of these is Emacs, which is suggested by the documentation. Additionally, two attempts have been made at more user friendly interfaces, both targeted at being easier to use for undergraduates: DrACuLa and ACL2s.

I have had some prior experience working with ACL2 user interfaces in the form of Try ACL2 (at `http://tryacl2.org`), an experimental website that I built that gives a restricted, web-accessible REPL for ACL2 along with a simple tutorial to help the user learn some of the basics of ACL2. Through this project, I got some experience working with ACL2 programmatically along with some feedback on this method of interaction from the programming community. Try ACL2 is more of an introductory tool rather than a fully featured development environment. Proof Pad builds on this work towards the goal of a more comprehensive environment, bringing it in line with the tools mentioned above.

DrACuLa DrACuLa [6] is a plugin for DrRacket (formerly DrScheme) that acts as an IDE for ACL2. It is primarily intended for classroom use. Because DrACuLa is a DrRacket plugin, it has an excellent text editor for Lisp syntax. Its parentheses matching and auto-indentation are indispensable. DrRacket also has a history of pedagogic usage and, as such, is designed to be easy for students and inexperienced programmers to pick up and use [7]. DrACuLa also extends ACL2 with support for a modular style of programming inspired by the Scheme dialect that DrRacket uses. In this style, interfaces to modules are separated from the implementation of the modules. The modules also have contracts that implementations must uphold, which are mechanically verified by ACL2 [8].

In addition to supporting a sizable subset of ACL2's syntax and features, DrACuLa extends ACL2 with a set of "teachpacks" — short, easy-to-use libraries with specific functionality. One notable example is DoubleCheck, a QuickCheck-inspired library for running automated, randomized tests of ACL2 code. DoubleCheck provides both an accessible jumping-off point for students to start thinking in terms of verifiable properties before going straight to theorems, as well as a useful method of generating counterexamples for failed theorems. The alternative, reading ACL2's output for a failed proof attempt, is intimidating for many students [9].

However, a few aspects of DrACuLa pose problems for effective classroom use. Installation of DrACuLa is a complex process that involves acquisition and installation of three separate software components, interaction with the command line, and some potential pitfalls that are difficult to recover from, especially for beginning programmers. In order for this tool to be usable in courses at the University of Oklahoma, I have compiled and maintained a lengthy document for the installation process and investigated many potential ways to streamline the process, all to no avail. Installing the software tools is a frequent frustration in these classes.

Because DrACuLa executes ACL2 definitions in Scheme, only some of ACL2's functionality is available. For example, arrays and macros are not supported, and supporting them would likely involve a lot of work. This dual implementation can also lead to bugs where certain functions can be used in DrACuLa, but are rejected or incorrect in ACL2 (and vice versa). Additionally, this dual-implementation can lead to error messages that are hard to follow or track down in many cases, especially in the included teachpack libraries where there are separate ACL2 and Racket implementations of each function.

The ACL2 Sedan The ACL2 Sedan, or ACL2s, is an Eclipse plugin and set of additional features for ACL2 that seek to make ACL2 easier to use. Using the metaphor of a sedan, ACL2s intends to provide a simple, low-maintenance interface for ACL2 at the cost of high-level performance and customization. ACL2s includes some extensions of ACL2's functionality that add various useful features, including a method by which it can automatically attempt to generate counter-examples directly from theorems [10].

ACL2s has an automatic counterexample generation tool that works by looking directly at theorems to determine the types of data to try to bind to free variables in the theorem body. This means that ACL2s doesn't require a different syntax for specifying tests and data generators. This approach integrates more directly with the existing proof process, as opposed to the DoubleCheck approach of integrating proofs with an existing testing method [11].

However, ACL2s also has some drawbacks. The entire UI for ACL2s is placed in a single toolbar consisting of nine buttons and a single top-level menu. This is fine for Eclipse plugins that also re-use Eclipse's menu items and standard functionality, but, for the most part, ACL2s does not. When looking at a file, ACL2's status is shown only through colors, and no visual feedback is given for errors.

ACL2s requires that Eclipse be installed in a non-standard path on Windows systems. The initial placement of the REPL (which is treated as a type of file) is in a separate tab, which the user must flip back and forth between to use, instead of the docked window arrangement shown on the website. The default mode that ACL2s users are put into is called "Bare Bones" mode, where common macros such as addition, subtraction, and equality testing are undefined, with no clear indication of how to change that.

Modern development environments frequently provide complex, language specific syntax highlighting, that takes into account different categories and types of keywords and built-in functionality, data types, and comments. ACL2s, however, does not; syntax highlighting simply consists of one color for parentheses, one for code, and one for comments. This makes typos difficult to spot and built-in functionality difficult to recall (e.g. is it `equal` or `equals`?).

1.3 Pedagogic IDEs

BlueJ BlueJ is a Java IDE developed by Michael Kölling and others with teaching as a primary goal. BlueJ experiments with several aspects of the standard IDE, while also seeking to simplify the steep learning curve of professional IDEs of then and now [12].

From a pedagogic perspective, Kölling and Rosenberg outline eight development guidelines for BlueJ and, more generally, for teaching Java. I think several of these guidelines apply effectively to ACL2 and Proof Pad, and I would like to see Proof Pad accommodate the teaching style they espouse. Of particular note to ACL2 are guidelines 2 ("Don't start with a blank screen"), 3 ("Read code"), and 5 ("Don't start with 'main'"). Overall, the relevant advice is to ease new programmers into the subject matter without overwhelming them with features and possibilities. Creating a totally new project from scratch can be intimidating for novice programmers, and getting it to run as a standalone program, from input to output with no intermediate, is frustrating. Instead, the authors argue, students should start with existing code and existing projects to give them a sense of what to emulate and to make the tasks both more straightforward and rewarding. The features discussed in subsection 6.1 try to allow the instructor to take advantage of these guidelines [13].

DrRacket DrRacket (formerly DrScheme) is a popular pedagogic IDE for the Scheme programming language. DrRacket started as a tool for teaching the introductory programming courses at Rice University. The team working on it sought to rectify several pedagogic flaws with standard Scheme, such as the too-forgiving Scheme syntax and the problems of redefining functions in the REPL. The design settled on a large definitions area which could be compiled and then run in the REPL, which was intended to be more stateless [7].

2 Design

2.1 Goals and Constraints

When designing Proof Pad, I started by documenting several goals and constraints intended to make Proof Pad an effective, modern development environment. These goals are based heavily on the results of a questionnaire completed by students in the software engineering course at the University of Oklahoma. The questionnaire posed three free-form questions, soliciting up to three answers to each. The results of the questionnaire were reviewed by Rex Page, Allen Smith, and myself as part of ongoing work in a project investigating pedagogic use of ACL2. As part of this review, we collected together responses we saw as duplicates, so that they could be prioritized.

Primary complaints about DrACuLa included the lack of a working runtime debugger/program stepper, the poor quality of runtime and syntax error messages, and the slow start-up time. Responses praised DrACuLa's parentheses matcher, property based testing, and integration with ACL2. Responders also expressed a desire for detection of syntax errors while typing [14]. Overall, the results revealed several areas of the existing platform, DrACuLa, that seemed that they could be improved upon by taking a new approach. Many of these elements could not be easily addressed in the DrRacket framework.

User Interface Inasmuch as is possible, Proof Pad should have a feature set inspired by industry standard IDEs such as Eclipse and Visual Studio. Professional and pedagogic IDEs frequently include syntax highlighting, automatic indentation, and bracket matching features to facilitate entering programs that are free of syntax errors. These features are a must, especially to students who are used to them in other environments.

Performance Having a fast start-up time is an important part of the user's perception of the speed and performance of the program overall, and tends to leave a more lasting impression than other performance measures. In the development of Proof Pad, I paid particular attention to this metric.

Cross platform Having an application that both runs on the user's operating system of choice and that also fits with their expectations for other applications on that platform is important to giving the user a good first impression of the software. Since ACL2 is already available on Windows, Mac, and Linux, Proof Pad should (and does) provide comparable cross-platform support. Additionally, even at the University of Oklahoma, which requires Windows for certain classes, a survey of Applied Logic (one of the target courses for Proof Pad) students' homework submissions determined that 36% of students use Macs, and 59% use Windows (the remaining 5% used a computer lab).

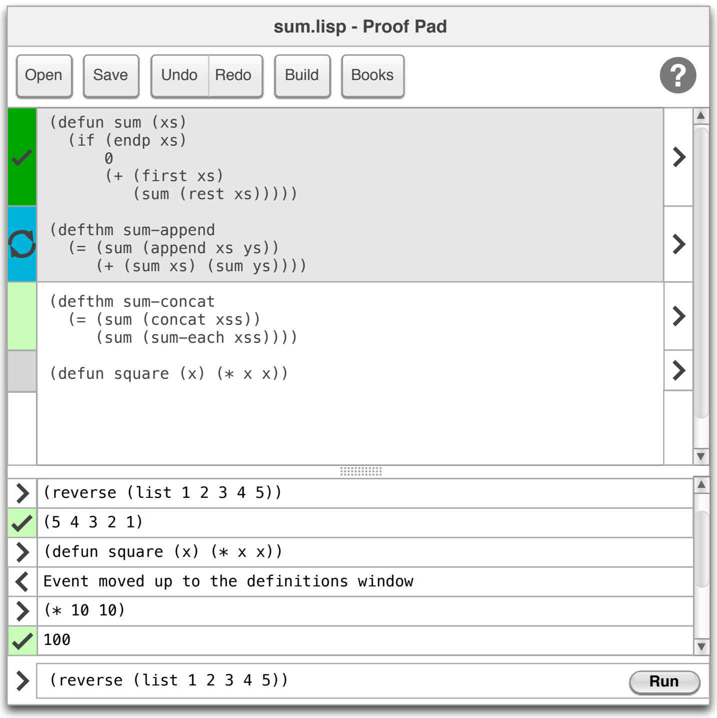
As much as possible, the tool should stay consistent with the platform it is running on, both graphically and interactively. As such, I worked towards compliance with the Windows User Experience Interaction Guidelines[1], the Mac OS X Human Interface Guidelines[2], and the GNOME Human Interface Guidelines[3].

User-focused process I sought feedback from students and other ACL2 users at all stages of development, in both formal and informal contexts. User testing is an essential part of creating a high quality, usable application. I want Proof Pad to be as free of usability problems as I can make it, to allow users to focus on their code, not the interface. I used a combination of one-on-one testing with individual students completing tasks that I designed and more broad-scale testing by using the tool in a classroom

[1] `http://msdn.microsoft.com/en-us/library/windows/desktop/aa511258.aspx`
[2] `http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines`
[3] `http://developer.gnome.org/hig-book/3.0/`

setting in two semester-long courses, from which I collected more aggregate and quantitative data using a questionnaire.

2.2 User Interface Components



```
(defun sum (xs)
  (if (endp xs)
      0
      (+ (first xs)
         (sum (rest xs)))))

(defthm sum-append
  (= (sum (append xs ys))
     (+ (sum xs) (sum ys))))

(defthm sum-concat
  (= (sum (concat xss))
     (sum (sum-each xss))))

(defun square (x) (* x x))
```

```
(reverse (list 1 2 3 4 5))
(5 4 3 2 1)
(defun square (x) (* x x))
Event moved up to the definitions window
(* 10 10)
100
```

```
(reverse (list 1 2 3 4 5))
```

Fig. 1. A wireframe of the design of the main Proof Pad window. A screenshot of the most recent version is shown in Figure 3.

The working area of Proof Pad is logically split into a few different areas. A wireframe of the main window is shown in Figure 1. The large area on top where most of the code editing occurs is called the definitions area, since the primary use for this part of the workflow is to define functions and theorems. To the left of the definitions area is the proof bar, which displays and allows manipulation of the state of ACL2 with respect to the definitions. Below these items is the read-eval-print loop, an interactive console where functions can be invoked to test or demonstrate their functionality.

Definitions Area The definitions area is where files are edited. Functions, properties, and theorems are defined in this area. The space is visually divided into areas that denote the state of ACL2: the Proof Bar (see section 2.2), the actual syntax-highlighted text area, and a bar to the right that allows the user to view the output of ACL2 for specific items.

The two bars to the left and right visually segment the definitions area based on the top-level events or function calls in the definitions area. This segmentation is further emphasized if part of the definitions have been admitted to ACL2, which renders them with a grey background to indicate that they are not editable and with a dark line below them to separate admitted and unadmitted items.

The syntax highlighting has been carefully tailored to ACL2. For example, ACL2 makes a fundamental distinction between regular functions (such as `equals`, `cons`, and `append`), and events, which modify the ACL2 state (such as `defun` and `defthm`). Proof Pad emphasises this difference by displaying the two types in different colors.

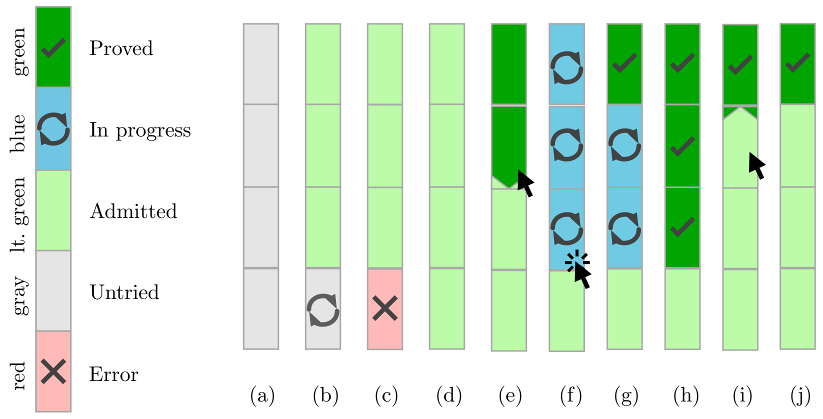


Fig. 2. Diagram showing how the proof bar responds to user interaction in some routine scenarios as a series of state transitions. (a) Initial state. (b) Three expressions automatically admitted. (c) Fourth expression has an error. (d) The error has been corrected. (e) User hovers over the bar. (f) User clicks the bar. Admission begins. (g) One expression has been admitted successfully to the logic. (h) All three requested expressions admitted successfully. (i) User hovers over second entry. (j) User has unadmitted two expressions.

Proof Bar Proof Pad introduces a user interface element for displaying and maintaining the status of the current document with relation to ACL2. Typical ACL2 workflow involves typing in a definition or theorem, attempting to admit it to ACL2, and either responding to errors or continuing with the next definition. Additionally, users occasionally need to return to a previously admitted expression and modify it to accommodate a case or error that they had not previously seen. Figure 2 shows some of the interactions the proof bar supports.

In order to accommodate these use cases, the proof bar responds to clicks in one of two ways:

1. If the click is to the left of an unadmitted expression, the proof bar queues all of the unadmitted top-level expressions down through (and including) the selected expression for admission. As ACL2 admits each one, the status changes from in progress to proven or failed.
2. If the click is to the left of an admitted expression, the proof bar undoes all of the admissions that changed the global state up through and including the selected expression. Undoing is done through the undo feature of ACL2, which has a fast response time.

Hovering over the proof bar previews the action that will be taken if the user clicks.

At all times, the proof bar indicates the current status of each top-level expression in the definitions area. There are five different statuses that are represented using different colors and symbols.

Read-eval-print loop The read-eval-print loop (REPL) is a common feature of Lisp-like languages. Formulas are typed at a prompt and executed in the context of the current set of definitions. A REPL is a popular and effective part of other pedagogic IDEs, such as DrRacket [7] and DrJava [15].

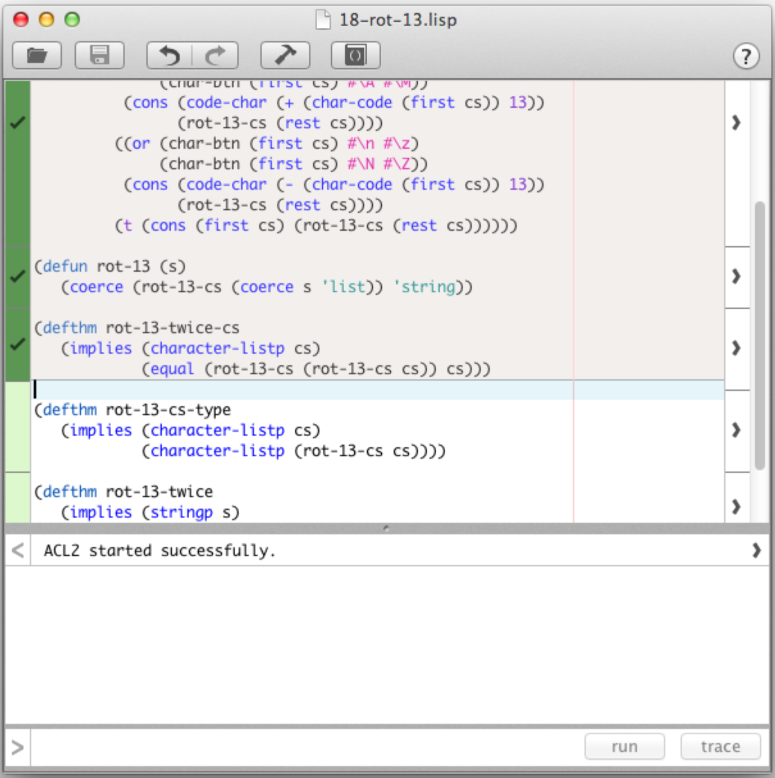
One issue with the traditional REPL that the developers of DrRacket discovered is that novice users often have trouble keeping track of stateful changes to the environment that they make using the REPL. For instance, a student might fix a bug by redefining a function in the REPL, but then forget to incorporate that change back into the definitions area. This can lead to bugs that were thought to be fixed returning later [7]. Proof Pad addresses this problem by keeping definitions consistent with updates to the REPL. All functions in ACL2 are divided into two groups: events, such as `defun` or `defthm`, which modify the global state; and functions, such as `max` or `first`, which do not modify the state of the system, but just return their result. Proof Pad automatically moves event formulas to the definitions area without passing them through ACL2, sidestepping the problem of state changes in the REPL.

Proof Pad's REPL has some unique enhancements over typical REPLs. For one, Proof Pad's REPL, like other parts of the UI, summarizes ACL2 responses to events, while still allowing the user to click a disclosure arrow in the UI to view the full ACL2 output.

Results window When Proof Pad encounters an error, or when the user clicks on one of the disclosure arrows either in the REPL or in the definitions area, the results window appears to the right of the main window. The format of the results window depends on the type of data being displayed. In most instances, it consists of a color-coded summary of error, warning, and success messages that Proof Pad has detected in ACL2's output, followed by a text area containing the raw ACL2 output.

3 Implementation

In total, Proof Pad comprises approximately 11,000 lines of Java code developed over a period of 13 months. Figure 3 shows a timeline for the project. The project has expanded significantly as it progressed. The full source code can be found at `http://github.com/calebegg/proof-pad`, and is available under the GPLv3 license.



```
            (char-btn (first cs) #\A #\M))
             (cons (code-char (+ (char-code (first cs)) 13))
                   (rot-13-cs (rest cs))))
            ((or (char-btn (first cs) #\n #\z)
                 (char-btn (first cs) #\N #\Z))
             (cons (code-char (- (char-code (first cs)) 13))
                   (rot-13-cs (rest cs))))
            (t (cons (first cs) (rot-13-cs (rest cs))))))

(defun rot-13 (s)
   (coerce (rot-13-cs (coerce s 'list)) 'string))

(defthm rot-13-twice-cs
   (implies (character-listp cs)
            (equal (rot-13-cs (rot-13-cs cs)) cs)))

(defthm rot-13-cs-type
   (implies (character-listp cs)
            (character-listp (rot-13-cs cs))))

(defthm rot-13-twice
   (implies (stringp s)
```

```
ACL2 started successfully.
```

Fig. 3. The main Proof Pad window, showing a file that is being actively worked on.

3.1 Code editor and syntax highlighting

Proof Pad uses the open-source Java code editor `RSyntaxTextArea` for the definitions pane and for the input to the REPL. `RSyntaxTextArea` makes certain familiar parts of code editing, such as syntax highlighting, easier to implement. I selected

March 2012	Began work on prototype	7,400 lines of code
April 27 2012	First user evaluation	8,800
Fall 2012	Trial use in logic course	9,800
December 2012	Survey of students in logic course	
Spring 2013	Trial use in logic course	10,900
March 2013	Second evaluation	

Fig. 4. Timeline showing some of the major events in the development of Proof Pad.

`RSyntaxTextArea` primarily for its speed and simplicity. The syntax highlighting parser works with a JFlex lexical analyzer generator. JFlex generates a minimal DFA from a regular expression based lexer definition so that the lexer code it generates is performant. As part of tailoring the syntax highlighting to ACL2, The JFlex lexer provided for Lisp needed to be heavily modified to make it aware of ACL2's keywords and some specific aspects of ACL2's primitive data types.

In addition to the syntax highlighting lexer, Proof Pad also has a slower, simple parser that runs after a few seconds of inactivity and scans the user's code, using the lexer's tokens, looking for several types of simple syntax errors, such as undefined functions or variables, too many or too few arguments for a built-in function or macro, or incorrect syntax for a built-in macro like `let` or `defthm`.

Proof Pad's auto-indentation feature watches for the user to enter a newline and determines the proper level of indentation using a full trace of the previous indent amounts and parentheses depth, taking into account specific indentation patterns for certain built-in macros. The user can also re-indent sections of code.

4 Evaluation

4.1 First user evaluation

Participants in the first Proof Pad evaluation came from The University of Oklahoma's software engineering course. This is the second semester of the capstone course for Computer Science, so many of the participants were seniors. Furthermore, because the course is taught using ACL2 in the DrACuLa environment, the participants already had a moderate amount of experience using ACL2, but with a different IDE. This made it possible to ask students participating in the evaluation to perform sophisticated evaluation tasks.

Some studies of usability testing have found that for qualitative usability studies like this one, small numbers of participants (four or five) are nearly as effective at finding most usability problems in an application as large numbers [16]. Five students participated in this initial evaluation.

The evaluation consisted of four tasks, each taking at most five minutes, followed by ten minutes allotted to discuss overall impressions of the tool, for a total of 30 minutes per student. Evaluations were performed individually with me on a Macbook Air, using either a Mac-like or Windows-like (with Windows-style keybindings and menus) build of Proof Pad. All students chose the Windows-style build.

For the first two tasks, the task was presented verbally alongside a Proof Pad window populated with some code. In the first task, the participant was asked to admit and run a provided function. In the second, a file that has three errors was shown, and the participant was asked to identify and fix any of them that they could, and then run the function.

For the third and fourth task, instructions were provided both verbally and on screen, along with an empty Proof Pad window for the student to work in. The first of these tasks asked them to write a function that computes the product of a list; one example is provided. The second task asks them to write and admit/run a theorem or property (DoubleCheck test) to demonstrate that `(prod (append xs ys))` = `(* (prod xs) (prod ys))`, using either their `prod` function from the previous task, or a provided one if they did not complete the task.

From this first evaluation, I gained much valuable feedback. Nearly all participants had trouble discovering the use and purpose of the Proof Bar. Several participants expressed a desire for some of the standard features of IDEs they've used before, such as line numbers, code folding, automatic insertion of parentheses, automatic saving, options to automatically fix highlighted errors, etc. In addition to this higher-level feedback, I also made several small usability changes to the application based on more minor problems experienced by only one or two participants.

4.2 Use in logic courses (Fall 2012 and Spring 2013)

Proof Pad was used as the primary software tool in the Fall 2012 section of Applied Logic and the Spring 2013 section of How Computers Work[5], a cross-listed honors course, at the University of Oklahoma. I provided support for the software throughout these two semesters, and the students completed several assignments using it. At the end of the semester, they were given an anonymous survey where they were asked to rate their confidence using certain parts of Proof Pad on a five-point scale, to supply (up to) three good things, bad things, and desired features for Proof Pad, and, finally, to rate their overall experience with Proof Pad on a five point scale. The quantitative questions show that students were reasonably comfortable with writing and managing their code, but not very confident in their ability to interpret and respond to error messages. The free-form questions also show that error messages were difficult to understand and documentation was hard to find [14].

4.3 Second evaluation

Another evaluation of Proof Pad took place in April of 2013. Participants were recruited from the University of Oklahoma's second-level introductory programming course as well as the Data Structures course. They were given an IDE-agnostic training session for the basics of ACL2, and then performed tasks one-on-one using either Proof Pad or DrACuLa (randomly selected). Seven students participated in the study; three used DrACuLa and four used Proof Pad. The tasks used to evaluate the two IDEs were very similar to the tasks used in the first user study. Some of them were modified to better fit with the participants' knowledge and skill level and to try to avoid bias in favor of one or the other of the two IDEs.

For each participant, I measured the amount of time that they took to complete the task. Participants were only given a maximum of five minutes, so after that time I considered the attempt unsuccessful. Table 1 shows the average time taken and the precise success rate for each task.

		Success rate		**Average time taken**	
		Proof Pad	**DrACuLa**	**Proof Pad**	**DrACuLa**
Task 1	Run in REPL	4/4	0/3	1:56	—
	Admit theorem	3/4	0/3	3:24	—
Task 2		4/4	2/3	2:50	1:14
Task 3	Variable name	4/4	2/3	1:03	0:57
	Swapped args	4/4	2/3	2:48	2:07
	Base case	3/4	2/3	3:25	1:33
Task 4		2/4	1/3	3:12	2:30
Task 5		4/4	2/3	3:10	1:44

Table 1. Some of the quantitative results from the second evaluation of Proof Pad and DrACuLa.

With only seven participants, it is hard to draw statistically significant conclusions. Students seemed to perform the tasks reasonably well with Proof Pad, and had a higher success rate, especially on the first task of finding out on their own how to run and admit functions. However, those who were successful with tasks in DrACuLa were able to complete their tasks consistently faster. More detailed analysis with more participants might reveal more interesting patterns. I was also able to identify and take note of six minor usability issues with Proof Pad.

5 Conclusion

As a software tool for the classroom, Proof Pad has been used successfully as a primary tool in two different sophomore level classes and as a secondary tool in one senior level class. It's available now at `http://proofpad.org/` for download, and will be used in at least one more class at the University of Oklahoma. As the project's source code is available, it is my hope that any features that are thought to be helpful could be more easily borrowed by other, similar projects. A primary goal of the project was to carefully evaluate how Proof Pad met student expectations and goals at all stages, and to respond effectively to student feedback. I think this had led to a high quality end product that's easier to learn and become familiar with.

6 Future work

6.1 `.proofpad` files

A planned feature to better support the pedagogic goals of Proof Pad is `.proofpad` files. These files would be created by instructors and would contain some unfinished code with missing functionality along with tests to determine whether the functionality has been implemented correctly. Proof Pad would divide these files up into sections that are and are not editable. The student's goal would be to make the entire file admit to ACL2 by going through their own code in the middle.

Through this mechanism, the instructor can easily work up in difficulty from simple tasks like modifying existing functions to cause failing tests to pass, all the way to advanced tasks like steering ACL2 to solve a complex theorem by adding lemmas to the logical world [14].

7 Acknowledgements

Rex Page is my Master's thesis advisor, under whom I've done this work. His support and feedback have been invaluable. Peter Reid provided feedback and several patches for issues he encountered early in the project. Matthew Kaney created the icon for Proof Pad, selected the color scheme for the proof bar, and gave feedback on other visual elements of the design. An early discussion about Proof Pad with Ruben Gamboa led to the idea for the `.proofpad` files. Proof Pad uses some open source components; in particular, the previously-discussed `RSyntaxTextArea` (written and maintained by Robert Futrell), and, of course, ACL2 itself. The compiled and certified binaries of ACL2 that Proof Pad uses were created and provided by the ACL2s project. Parts of DoubleCheck and several other DrACuLa libraries, written by Carl Eastlund and others, are included with Proof Pad for compatibility with files that use these libraries.

References

1. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell, MA, USA (2000)
2. Eastlund, C., Vaillancourt, D., Felleisen, M.: ACL2 for freshmen: First experiences. In: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, ACM Press (2007) 200–211
3. Page, R.: Engineering software correctness. Journal of Functional Programming **17** (November 2007) 675–686
4. Page, R.: Property-based testing and verification: a catalog of classroom examples. In: Proceedings of the 2011 Symposium on Implemenation and Application of Functional Languages, Lawrence, KS (October 2011) 134–147
5. Page, R., Gamboa, R.: How computers work: Computational thinking for everyone. In: Proceedings of the First International Workshop on Trends in Functional Programming in Education, St Andrews, UK (June 2012)
6. Vaillancourt, D., Page, R., Felleisen, M.: ACL2 in DrScheme. In: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications, New York, NY, USA, ACM (2006) 107–116

7. Findler, R., Flanagan, C., Flatt, M., Krishnamurthi, S., Felleisen, M.: DrScheme: A pedagogic programming environment for scheme. In Glaser, H., Hartel, P., Kuchen, H., eds.: Programming Languages: Implementations, Logics, and Programs. Volume 1292 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 369–388
8. Eastlund, C., Felleisen, M.: Toward a practical module system for ACL2. Practical Aspects of Declarative Languages (2009) 46–60
9. Eastlund, C.: DoubleCheck your theorems. In: Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications, New York, NY, USA, ACM (2009) 42–46
10. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: "The ACL2 Sedan". In: Proceedings of the 7th Workshop on User Interfaces for Theorem Provers. Volume 174 of Electronic Notes in Theoretical Computer Science. (2007) 3 – 18
11. Chamarthi, H.R., Dillinger, P.C., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving. In Hardin, D., Schmaltz, J., eds.: Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications,. Volume 70 of Electronic Proceedings in Theoretical Computer Science., Open Publishing Association (2011) 4–19
12. Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The BlueJ system and its pedagogy. Computer Science Education **13**(4) (2003) 249–268
13. Kölling, M., Rosenberg, J.: Guidelines for teaching object orientation with Java. ACM SIGCSE Bulletin **33**(3) (2001) 33–36
14. Eggensperger, C.: Proof pad: A modern development environment for ACL2. In: Proceedings of the Eleventh International Workshop on the ACL2 Theorem Prover and its Applications. Electronic Proceedings in Theoretical Computer Science (2013) To appear.
15. Allen, E., Cartwright, R., Stoler, B.: DrJava: a lightweight pedagogic environment for Java. In: Proceedings of the 33rd SIGCSE technical symposium on Computer science education, New York, NY, USA, ACM (2002) 137–141
16. Virzi, R.: Refining the test phase of usability evaluation: How many subjects is enough? Human Factors: The Journal of the Human Factors and Ergonomics Society **34**(4) (1992) 457–468