# A User-friendly Interface for a Lightweight Verification System[1]

Andrei Lapets[2]

*Computer Science Department*
*Boston University*
*Boston, USA*

Assaf Kfoury[3]

*Computer Science Department*
*Boston University*
*Boston, USA*

**Abstract**

User-friendly interfaces can play an important role in bringing the benefits of a machine-readable representation of formal arguments to a wider audience. The AARTIFACT system is an easy-to-use lightweight verifier for formal arguments that involve logical and algebraic manipulations of common mathematical concepts. The system provides validation capabilities by utilizing a database of propositions governing common mathematical concepts. The AARTIFACT system's multi-faceted interactive user interface combines several approaches to user-friendly interface design: (1) a familiar and natural syntax based on existing conventions in mathematical practice, (2) a real-time keyword-based lookup mechanism for interactive, context-sensitive discovery of the syntactic idioms and semantic concepts found in the system's database of propositions, and (3) immediate validation feedback in the form of reformatted raw input. The system's natural syntax and database of propositions allow it to meet a user's expectations in the formal reasoning scenarios for which it is intended. The real-time keyword-based lookup mechanism and validation feedback allow the system to teach the user about its capabilities and limitations in an immediate, interactive, and context-aware manner.

*Keywords:* formal verification, user interfaces

[1] This material is based in part upon work supported by the National Science Foundation under Grant Numbers 0820138 and 0720604. 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.

[2] Email: lapets@bu.edu

[3] Email: kfoury@bu.edu

*This paper is electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

## 1 Introduction

User-friendly interfaces can play an important role in bringing the benefits of adopting a machine-readable representation of formal arguments to a wider audience. There exist many such benefits both in mathematical instruction and in research efforts involving mathematical rigor. These include reusability, automatic evaluation of examples, and the opportunity to employ machine verification. Machine verification can offer anything from detection of basic errors, such as the presence of unbound variables or type mismatches, to full confidence in an argument because it is consistently constructed using the fundamental principles of a particular mathematical logic. There exist a variety of such machine verification systems, and some of these have been surveyed and compared along a variety of dimensions [33].

Until more recently, however, user interface design has not been a major focus of the formal verification community. Earlier efforts make claims that verification systems allow "human-readable" representations of formal arguments [32,25,1]. However, conventions governing the concrete syntax for representing even some basic and ubiquitous formal constructs (e.g. notation for representing vector concatenation, or for representing graphs) are not consistent. Furthermore, in order to be of practical use, verification systems must incorporate very large libraries of definitions and propositions. Consequently, even if a verification system has a simple core syntax, an expert user that wishes to employ it must first become familiar with any libraries that might be pertinent to the task at hand. In this way, the true syntax (consisting of syntactic idioms corresponding to library content) of such systems can still be obscure even to expert users. The issue of teaching users interactively about system capabilities and limitations (including libraries of results users might need to employ) has not yet been addressed sufficiently well.

The purpose of a user interface is two-fold. First, it must meet the user's expectations by providing an abstraction of the system that corresponds to the user's intuition and experience. Second, it must make clear what is expected of the user in a way that is immediate, interactive, and context-aware. We present our user interface design for the AARTIFACT system,[4] a lightweight verification system for formal arguments that involve manipulation of common mathematical concepts. The interface has a multi-faceted design that works towards meeting these two criteria for a user-friendly interface. It incorporates three approaches: a familiar and natural syntax based on existing conventions in the practice of formal reasoning, a keyword-based lookup mechanism for discovery of supported syntactic idioms and semantic concepts, and feedback in the form of reformatted raw input.

[4] An interactive demonstration is available at http://www.aartifact.org.

## 2 Motivation and Background

Introduce $P, m$.
Assume $P$ is a finite set, $P$ is non-empty, and $P \subset \mathbb{N}$.
Assume for all $n \in \mathbb{N}$, if $n$ is prime then $n \in P$.

Assume $m = P_0 \cdot \ldots \cdot P_{|P|-1}$.
Assert $m \in \mathbb{N}$.

Assert for any $p \in \mathbb{N}$,
    if $p$ is a prime factor of $m + 1$ then
        $p$ is not a factor of $m$,

        $p$ is prime,
        $p \in P$,
        $p$ is a factor of $m$,
        there is a contradiction.

Fig. 1. An example of a proof of the infinitude of primes authored using the AARTIFACT system.

Even if one considers a small collection of mathematical concepts, a practicing mathematician might use a large number and a great variety of syntactic idioms to refer to the predicates and operators that relate to them. To illustrate this, Figure 1 presents a very short proof of the infinitude of primes. This proof contains explicit references to finite sets, natural numbers, prime numbers, products, and factors. It also contains many implicit references to the properties of these concepts, and to the relationships between them. In order to accept arguments written in this manner as input, a user interface must be both flexible and robust. In this section, we briefly review the relevant approaches (some of which are based on those adopted in related work) that can be useful in designing such an interface. Section 3 describes how each of these was employed in the design of the user interface for the AARTIFACT system. For the purposes of discussion, we adopt the following terminology: a user constructs an *argument* in some form (possibly with the help of the user interface), and the interface then provides *feedback* to the user of the argument's validity (in the form of error messages, highlights, and so on).

### *2.1 Natural Syntax and Concrete Representation*

Any system that aims to support the kind of formal reasoning activity users employ in constructing a proof such as the one in Figure 1 must at least

provide a natural syntax that corresponds to the conventions that prevail in the target community of users. The designers of Scunak mathematical assistant [5] echo this in positing a need for "naturality" in a system's concrete representation. The system must provide at least some familiar but simple syntactic constructs for assembling logical arguments (i.e. conjunction, disjunction, quantification). Furthermore, even if the system incorporates an extensive library containing many concepts, properties, and relationships that a user may want to employ, the system must allow the user to employ many of these without explicitly referencing them (i.e. it must *not require* the user to name the results from a library when the user wishes to employ them). The designers of the Scunak system [5] refer to this as "[retrievability] ... by content rather than by name." Likewise, the designers of the `Tutch` system posit that an "explicit reference [to an inference rule] to [humans] interrupts rather than supports the flow of reasoning" [1].

### *2.2 Search and Automatic Keyword Lookup for Syntactic Idioms*

Syntax is a means of communication, and a simple and natural formal syntax is useful because it provides a means that can be learned quickly for encoding formal arguments. However, this simple syntax must then be used to represent a large library of operators, predicates, and even syntactic idioms. It is necessary to both store all these conventions in some sort of database, and to expose them to a user without requiring that they spend time and effort reading documentation or browsing a library. Thus, while an indexed database of syntactic idioms (or, more generally, typed terms [20], or logical definitions and theorems [8]) is a natural starting point, real-time keyword-based lookup techniques for programming environments [10,18] suggest a means for further improving the usability of a system. The system's interface can interactively inform the user about any relevant syntactic idioms and concepts found in the library by interactively displaying references and examples based on the text the user is typing in her argument.

### *2.3 Feedback about Logical Validity*

Feedback provided to the user about an argument's validity can include notifications about syntax errors and unbound variables, as well as about assertions that are unverifiable or false with respect to some logic. There are three important characteristics of this feedback that can contribute to the system's usability and flexibility: the legibility and understandability of the feedback (e.g. precise indication of the location of errors), the option to easily select the kind of feedback the user desires (e.g. the validation technique [30] or logical system the user wishes to employ), and the speed with which the feedback can be provided (which may depend on the choice of validation procedure).

## 3 Interface Design for a Lightweight Verifier

We describe in more detail the overall design and individual components of the interface for the AARTIFACT lightweight verification system. Figures 2 and 3 illustrate the user interface from the user's perspective. The user submits a formal *argument* represented using concrete syntax. If the "library" or "syntax" tab is selected, real-time hints for supported syntax are provided based on the text surrounding the cursor as the user types. The user can also select a logical system and click "verify" to produce feedback that replicates the input as HTML with color highlights indicating valid and invalid portions of the argument (with blue and red, respectively).

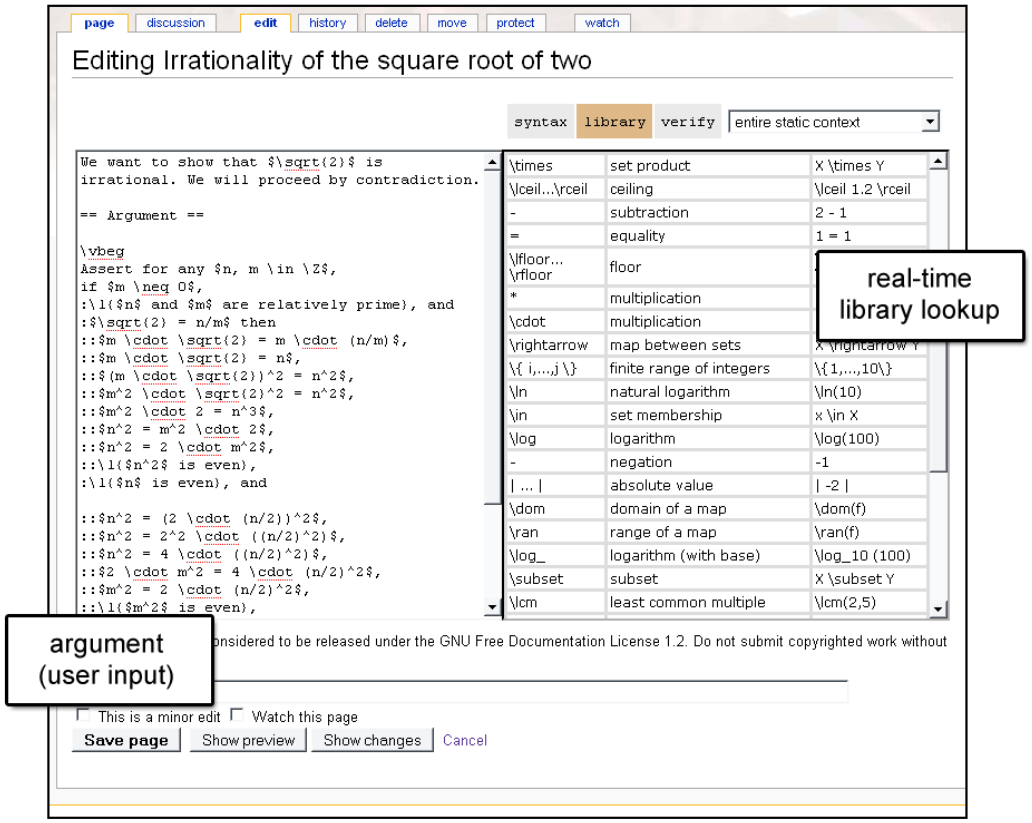


Fig. 2. Screen capture of the user interface in "library" mode.

Figure 4 illustrates the overall organization of the various components of the system, and how they behave in practice. An expert-managed database contains a library of syntactic constructs and propositions. This database is compiled into a client-side JavaScript application for syntax lookup, and a server-side executable that can perform formal verification. This ensures that only the server must be trusted to perform verification correctly, while the computational burden of providing syntax lookup is carried by the client ma-

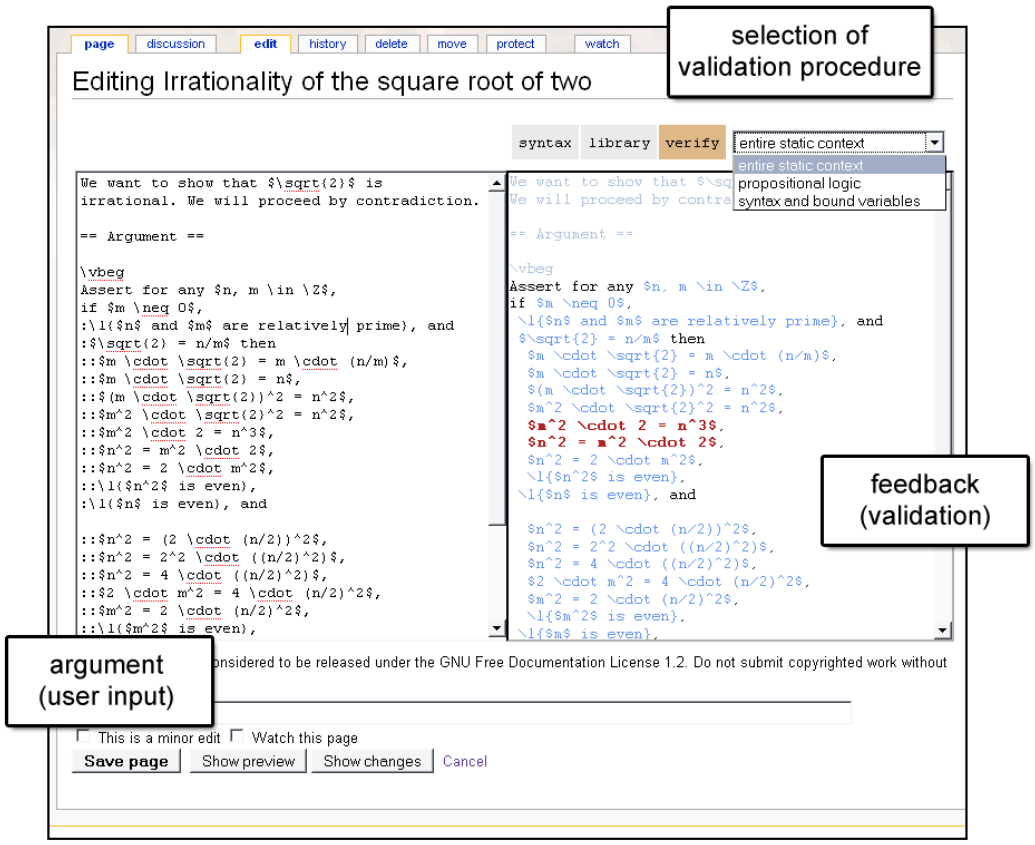


Fig. 3. Screen capture of the user interface in "verification" mode.

chine. The server sends the JavaScript application to the client when the web interface illustrated in Figures 2 and 3 first loads. Users then author arguments within their own browser with the help of the JavaScript application, and have the option of submitting their arguments to the server for validation at any time with the click of a button.

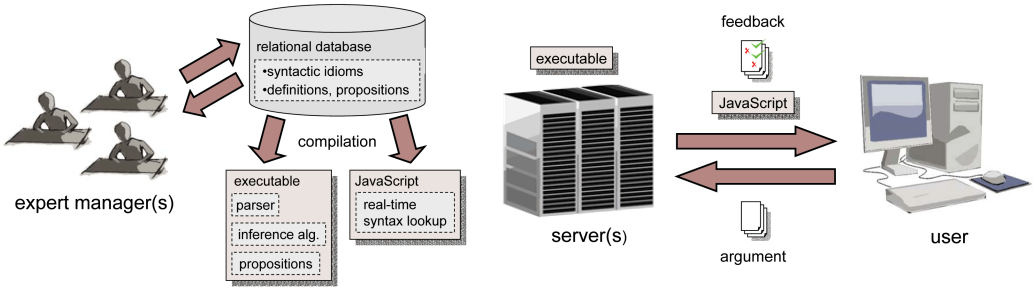


Fig. 4. Overview of system components and operation.

### *3.1 Concrete Syntax for Arguments*

The concrete syntax for arguments (listed in part in Figure 5) consists of English phrases, LaTeX markup, and MediaWiki markup. We denote by $\overline{x}$ a sequence of comma-separated variable identifiers. An argument consists of a sequence of *statements*. There are only *three* kinds of statements, and two of them (`Assume` and `Assert`) are very similar from the user's perspective. Each statement either introduces global variables, introduces an assumption, or represents an assertion about something that the user believes to be true. The syntax for logical expressions corresponds to typical English representations of logical operators within a higher-order logic. The two base cases for logical expressions are a mathematical expression in LaTeX syntax, and an English phrase predicate.

| | | | |
|---|---|---|---|
| statements | $S$ | ::= | Assume $E$ \| Assert $E$ \| Intro $\overline{x}$ |
| logical expressions | $E$ | ::= | `$` e `$` |
| | | \| | `\l{`$w_1$ $w_2$ ... $w_n$`}` |
| | | \| | $E_1$ iff $E_2$ |
| | | \| | $E_1$ implies $E_2$ |
| | | \| | $E_1$ and $E_2$ |
| | | \| | $E_1$ or $E_2$ |
| | | \| | it is not the case that $E$ |
| | | \| | for all `$` $\overline{x}$ `$`, e |
| | | \| | exists `$` $\overline{x}$ `$`, e |
| word or math expression | $w$ | ::= | English word \| `$` e `$` |
| mathematical expressions | $e$ | ::= | 1 \| 2 \| ... |
| | | \| | $x$ |
| | | \| | $e_1$ $e_2$ |
| | | \| | $(e)$ |
| | | \| | `\emptyset` |
| | | \| | $\{e\}$ |
| | | \| | $e_1, \ldots, e_n$ |
| | | \| | $e_1 + e_2$ |
| | | \| | ⋮ |

Fig. 5. Concrete syntax overview.

English phrases acting as predicates can have zero or more arguments. An English phrase predicate is represented within the abstract syntax (not presented in this paper but found in relevant reports [14]) using a list of words

(string literals without punctuation or spaces) and placeholders (which we denote [ ]). If the English phrase contains any mathematical arguments, the English phrase predicate is *applied* to a tuple of expressions representing the arguments. For example, the predicate found in the expression `\l{$p$ is a path in $G$}` is represented using the list

$$\{[\,], \text{is}, \text{a}, \text{path}, \text{in}, [\,]\},$$

and the entire expression is represented as

$$\{[\,], \text{is}, \text{a}, \text{path}, \text{in}, [\,]\}\ (p, G).$$

Mathematical expressions are represented using many typical LaTeX syntactic constructs. A collection of constants and operators (which is consistent with the basic commands found in existing LaTeX packages where possible) is supported. Some limited user extensibility is supported directly: users can define their own infix operators. More sophisticated extensions to the supported syntax require a custom build of the system.

It is the task of the parser to process the concrete syntax of the portion of an argument that is to be considered for verification. The parser for the concrete syntax is constructed in Haskell using the Parsec parser combinator library [17], which is expressive enough for constructing infinite lookahead parsers for general context-sensitive grammars. This library is simple to use and allows for a succinct parser implementation. The AARTIFACT parser performs without noticeable delay on all inputs on which it has been tested (the infinite lookahead capability is utilized at only a few points in the parser definition, such as to allow expert users to define their own infix operators). The overall approach to handling syntactic idioms using a context-sensitive parser is similar to the approach employed in the design of the parser for the Fortress programming language [26].

### *3.2 Library Access*

The current AARTIFACT library of supported propositions and definitions contains a collection of hundreds of entries. Each proposition deals with semantic concepts, properties they may have, and relationships that may hold between them. The following proposition represents a very simple example:

> "for any $x, y, z$,
> $x \in \mathbb{R},\ y \in \mathbb{R},\ z \in \mathbb{R},\ x < y,\ y < z$
> implies that
> $x < z$".

Many of these propositions simply state an equivalence between two forms of notation or syntax. They can be viewed as establishing a normal form for representing certain concepts or properties thereof. For example, the following proposition converts the typical notation for a set of integers in a finite range, "$\{x, \ldots, y\}$", into a predicate that is then used in other propositions about the properties of sets of integers in a finite range:

"for any $x, y$,
$x \in \mathbb{Z},\ y \in \mathbb{Z},\ x \leq y$
implies that
$\{x, \ldots, y\}$ is the set of integers ranging from $x$ to $y$".

The purpose of the library in its current incarnation is to support common algebraic manipulations and concepts in arithmetic and naive set theory. In particular, it includes: many common unary and binary operators on numbers, sets, and vectors; some common unary, binary, and ternary relations corresponding to properties of and relationships between numbers, sets, and vectors; and propositions that specify how operators preserve or affect these properties and relationships. As such, the current library does not yet have the breadth and depth to support reasoning about sophisticated concepts in a broad range of formal domains. It is our intention to continue expanding this library to accommodate the requirements of applications of the system.

Learning all the possible syntactic constructs and idioms for common concepts that the current library supports can be a time-consuming process for a user. Thus, a real-time keyword lookup system is integrated into the user interface to facilitate this process. Whenever a user is typing an argument, the text immediately surrounding the user's cursor is broken up into keywords, and these are then used to look up and present suggestions and examples of relevant syntactic constructs. Figure 2 illustrates how such suggestions look. If the "library" tab is selected, the panel on the right lists a variety of supported syntactic idioms corresponding to constants and predicates. If some collection of keywords produces a large number of results, the user may type more keywords to narrow these results. Facilities for informing the user of relevant related keywords are being developed as part of ongoing work.

Using these features, the system is able to inform the user of what is expected in a context-sensitive manner, and in doing so to establish a mode of communication with a user who may already know about the concepts she wishes to employ, but may not yet be familiar with the system's syntax or library. This is essential when the user wishes to employ concepts and notations that are not necessarily consistent within the community. For example, the supported forms of notation for closed real number intervals might be $\{x \mid 0 \leq x \leq 10\}$ and $[0, 10]$, the notation for the set difference operator

might be \ or −, and the notation for concatenation of vectors might be · or ∘. Informing the user of these conventions within a context in which they are thinking about them saves time and provides an opportunity to learn the system's syntax within a relevant context. Even if the user is not familiar with any syntactic convention, she may temporarily type keywords related to the concept in question directly into the argument in order to receive information about supported notations for that concept.

This feature is implemented as a Javascript application that is compiled from the contents of the library. The JavaScript application is delivered to the user's browser whenever the web interface page is loaded, and the user's browser executes it. This approach makes it possible to provide instant feedback without burdening the server, which must process validation requests and generate feedback.

### *3.3 Validation Feedback*

The AARTIFACT web interface provides a means for selecting one of a (currently very small) collection of validation techniques. As illustrated in Figure 3, when validation is requested the raw ASCII text of an argument is processed and converted into HTML feedback in which colors are used to indicate both errors (e.g. unbound variables, unverifiable subexpressions in assertions) and verifiable assertions. This is accomplished by maintaining a data structure within the parser that couples the abstract syntax with the original concrete syntax. It is worth noting that while only the AARTIFACT verification executable is currently utilized, any other verification tool with a command-line interface that can accept ASCII input and can produce text or HTML output could be invoked using this interface.

## 4 Notes on Usability Evaluation

We have utilized [16] the AARTIFACT system in defining and reasoning about a compositional formalism for a typed domain-specific language [4]. The ability to implicitly invoke propositions dealing with the algebra of sets was essential in making this process manageable and in allowing the resulting proofs to be legible. This exercise also led to the discovery of a few minor errors and to the simplification of a few definitions in the compositional formalism.

The system has also been deployed for several formal reasoning assignments within two undergraduate courses:[5] an advanced undergraduate course on

[5] The courses in question were: the fall 2009 iteration of "Concepts of Programming Languages" and the spring 2010 iteration of "Geometric Algorithms". Both are required Computer Science curriculum courses for undergraduates within the Computer Science Department at the Boston University College of Arts and Sciences.

functional programming [15] and an introductory undergraduate course in linear algebra. The deployment in the class on functional programming was limited in scope and constituted only a "trial" run, while the deployment in the class on linear algebra constituted a small initial experiment. More extensive future experiments are planned; these will require the development of a cloud computing infrastructure that can ensure the reliable performance of the system in courses with many students.

In the linear algebra course, for the first homework assignment students were given eight simple statements to prove (these constituted a proof that $\mathbb{R}^2$ is a vector space given that $\mathbb{R}$ is a vector space). They were given only an example of a verified proof to guide them in their use of the aartifact system. No tutorial explaining the system's features was provided. Students could consult the lecture notes and textbook, which were written in a conventional manner without consideration for automated verification. Several students posed questions to staff using a mailing list; these questions were of a manageable scope and were answered with either an example or a clarification.

| assignment | #1 | #2 | #3 |
|---|---|---|---|
| submissions that were verifiable as fully correct | 7 | 7 | 13 |
| submissions with $\leq 20\%$ unverifiable/missing formulas | 4 | 3 | 1 |
| submissions with $> 20\%$ unverifiable/missing formulas | 2 | 2 | 1 |
| total number of submissions (from 16 students) | 13 | 12 | 15 |
| # of mailing list questions on using aartifact | 5 | 3 | 2 |

Fig. 6. Performance of students on assignments requiring automatically verifiable proofs in an undergraduate course on linear algebra.

The course had 16 enrolled students of all levels (1st-4th year undergraduates), and for this first assignment 13 attempted to submit proofs verified with aartifact. The assignment was completed with major errors by 2 students, with a few minor errors by 4 students, and without errors by the remaining 7 students. Subsequent assignments were of a similar scope and led to similar results, which are presented in Figure 6. That students were able to employ the system with minimal guidance (beyond the conventional definitions provided during lecture and in readings) demonstrates the usability benefits of meeting student expectations with the help of a natural syntax and an underlying library from which results can be invoked implicitly. That students were able to receive partial credit for incomplete proofs demonstrates one potentially useful aspect of employing a lightweight verification approach.

## 5 Related and Future Work

The aartifact syntax reflects the design principles of other formal verification systems such as `Tutch` [1] and Scunak [5]. The need for natural interfaces

in machine verification in general has been recognized by the designers of the `Tutch` proof checker [1], the Scunak mathematical assistant system [5], the ForTheL language and SAD proof assistant [30], the EPGY Theorem-Proving Environment [19], the ΩMEGA proof verifier [28], the ProveEasy system [6], in the work of Sieg and Cittadini [27], and in the work of Hallgren and Ranta [9]. To better serve users in engineering, mathematics, and the applied sciences, the Fortress programming language [2] incorporates common mathematical symbols and syntactic constructs into its syntax, and the designers are putting effort into assembling a flexible parser that simplifies user-directed expansion of the language syntax [26]. More widely, there exist other efforts to create interfaces and systems for practical formalization of mathematics. The MathLang project [12] is an extensive, long-term effort that aims to make natural language an input method for mathematical arguments and proofs.

The aartifact system's concrete syntax and parser can be improved further by adding support for additional syntactic constructs and idioms, and by providing more information within the HTML feedback (e.g. about the justification for verifiable assertions, and the counterexamples for false assertions). It may also be worthwhile to introduce input and output support for standards such as MathML [7].

Somewhat relevant work in providing search capabilities for a library of expressions has been done within the context of Haskell. Search facilities have been developed that allow users to retrieve and browse expressions within a context by their type [13], and there exists an online search tool called Hoogle for exploring the Haskell libraries [20]. The work of Hallgren and Ranta [9] presents a proof editor that uses a natural language parsing framework in conjunction with type checking to interactively help the user utilize supported formal and natural language syntactic constructs while authoring a verifiable proof. Matita [3] is a proof assistant the automation of which is heavily based on an integrated search engine. There has also been work on retrieval of library functions, and even automated construction of programming language code snippets, using collections of keywords [10,18]. This work suggests that our own future efforts can be directed into better integrating the real-time lookup functionality with validation capabilities. For example, the real-time lookup hints can actually provide suggestions for valid expressions that consist of variables that are within the scope of an assertion in an argument. More generally, it may be possible to represent some simple validation techniques (e.g. unbound variable detection) in their entirety as JavaScript applications.

Our work involves the creation of a web interface implemented using HTML and JavaScript. This strategy is similar to that employed in related work [11], though that work focuses on delivering (using only a web browser) the look, feel, and functionality of an existing proof assistant.

There is a variety of other tools for formal representation and machine ver-

ification of proofs, such as Coq [22], PVS [21], and Isabelle [23,24]. Many of these have been surveyed and compared along dimensions pertaining to usability [33]. These systems usually require that a user have some understanding of logic and formal systems before she can verify even the basic mathematical arguments we aim to support in our work. Interfaces for a system like Coq usually require the user to work within a rigid interactive framework and to assemble proof scripts that do not necessarily reflect the style of presentation employed by mathematics textbooks. More specifically, formal representation and verification systems whose designs share some of the motivations underlying our work include Isabelle/Isar [31] and Mizar [29]. In particular, Isabelle/Isar is designed to be relatively independent of any particular underlying logic, and both systems are designed with human readability in mind. There is some work in keyword-based lookup for Mizar [8], but it does not involve providing the user with real-time syntactic and semantic hints, and the searchable library is not implicitly integrated with the validation procedure.

## References


[1] Abel, A., B. Chang and F. Pfenning, *Human-readable machine-verifiable proofs for teaching constructive logic*, in: U. Egly, A. Fiedler, H. Horacek and S. Schmitt, editors, *PTP '01: IJCAR Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs*, Siena, Italy, 2001.

[2] Allen, E., D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr. and S. Tobin-Hochstadt, *The Fortress Language Specification Version 1.0* (2008).
URL http://research.sun.com/projects/plrg/fortress.pdf

[3] Asperti, A., C. S. Coen, E. Tassi and S. Zacchiroli, *User interaction with the matita proof assistant*, J. Autom. Reason. **39** (2007), pp. 109–139.

[4] Bestavros, A., A. Kfoury, A. Lapets and M. Ocean, *Safe Compositional Network Sketches: The Formal Framework*, in: *HSCC '10: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (in conjunction with CPSWEEK)*, Stockholm, Sweden, 2010, pp. 231–241.

[5] Brown, C. E., *Verifying and Invalidating Textbook Proofs using Scunak*, in: *MKM '06: Mathematical Knowledge Management*, Wokingham, England, 2006, pp. 110–123.

[6] Burstall, R. M., *Proveeasy: helping people learn to do proofs*, Electr. Notes Theor. Comput. Sci. **31** (2000), pp. 16–32.

[7] Buswell, S., S. Devitt, A. Diaz, P. Ion, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor and S. Watt, *Mathematical Markup Language (MathML) 1.01 Specification (Abstract)* (1999).
URL http://www.w3.org/TR/REC-MathML

[8] Cairns, P. and J. Gow, *Integrating searching and authoring in mizar*, J. Autom. Reason. **39** (2007), pp. 141–160.

[9] Hallgren, T. and A. Ranta, *An extensible proof text editor*, in: *LPAR '00: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning* (2000), pp. 70–84.

[10] Han, S., D. R. Wallace and R. C. Miller, *Code completion from abbreviated input*, in: *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (2009), pp. 332–343.

[11] Kaliszyk, C., *Web interfaces for proof assistants*, Electron. Notes Theor. Comput. Sci. **174** (2007), pp. 49–61.

[12] Kamareddine, F. and J. B. Wells, *Computerizing Mathematical Text with MathLang*, Electron. Notes Theor. Comput. Sci. **205** (2008), pp. 5–30.

[13] Katayama, S., *Library for systematic search for expressions*, in: *AIC '06: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications* (2006), pp. 381–387.

[14] Lapets, A., *Improving the accessibility of lightweight formal verification systems*, Technical Report BUCS-TR-2009-015, Computer Science Department, Boston University (2009).

[15] Lapets, A., *Lightweight Formal Verification in Classroom Instruction of Reasoning about Functional Code*, Technical Report BUCS-TR-2009-032, CS Dept., Boston University (2009).

[16] Lapets, A. and A. Kfoury, *Verification with Natural Contexts: Soundness of Safe Compositional Network Sketches*, Technical Report BUCS-TR-2009-030, CS Dept., Boston University (2009). URL http://www.cs.bu.edu/techreports/2009-030-verified-netsketch-soundness.ps.Z

[17] Leijen, D. and E. Meijer, *Parsec: Direct Style Monadic Parser Combinators for the Real World*, Technical Report UU-CS-2001-27, Departement of Computer Science, Universiteit Utrecht (2001).

[18] Little, G. and R. C. Miller, *Keyword programming in java*, in: *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (2007), pp. 84–93.

[19] McMath, D., M. Rozenfeld and R. Sommer, *A Computer Environment for Writing Ordinary Mathematical Proofs*, in: *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming* (2001), pp. 507–516.

[20] Mitchell, N., *Hoogle overview*, The Monad.Reader **12** (2008), pp. 27–35.

[21] Owre, S., J. M. Rushby, and N. Shankar, *PVS: A prototype verification system*, in: D. Kapur, editor, *CADE: Proceedings of the 11th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence **607** (1992), pp. 748–752.

[22] Parent-Vigouroux, C., *Verifying programs in the calculus of inductive constructions*, Formal Aspects of Computing **9** (1997), pp. 484–517.

[23] Paulson, L., "Isabelle: A Generic Theorem Prover," Springer, 1994.

[24] Paulson, L. C., *Generic automatic proof tools*, in: *Automated reasoning and its applications: essays in honor of Larry Wos*, MIT Press, Cambridge, MA, USA, 1997 pp. 23–47.

[25] Rudnicki, P., *An overview of the Mizar project*, in: *Proceedings of the 1992 Workshop on Types and Proofs for Programs*, 1992, pp. 311–332.

[26] Ryu, S., *Parsing fortress syntax*, in: *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (2009), pp. 76–84.

[27] Sieg, W. and S. Cittadini, *Normal natural deduction proofs (in non-classical logics)*, in: D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning*, Lecture Notes in Computer Science **2605** (2005), pp. 169–191.

[28] Siekmann, J. H., C. Benzmüller, A. Fiedler, A. Meier and M. Pollet, *Proof Development with OMEGA: sqrt(2) Is Irrational*, in: *LPAR*, 2002, pp. 367–387.

[29] Trybulec, A. and H. Blair, *Computer Assisted Reasoning with MIZAR*, in: *Proceedings of the 9th IJCAI*, Los Angeles, CA, USA, 1985, pp. 26–28.

[30] Verchinine, K., A. Lyaletski, A. Paskevich and A. Anisimov, *On Correctness of Mathematical Texts from a Logical and Practical Point of View*, in: *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics* (2008), pp. 583–598.

[31] Wenzel, M. and L. C. Paulson, *Isabelle/Isar*, in: F. Wiedijk, editor, *The Seventeen Provers of the World*, Lecture Notes in Computer Science **3600** (2006), pp. 41–49.

[32] Wenzel, M. M., "Isabelle/Isar - A versatile environment for human-readable formal proof documents," Ph.D. thesis, Institut für Informatik, Technische Universität München (2002).

[33] Wiedijk, F., *Comparing mathematical provers*, in: *MKM '03: Proceedings of the Second International Conference on Mathematical Knowledge Management* (2003), pp. 188–202.