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 to a wider audience the benefits of a machine-readable representation of formal arguments. 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 large 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 large database of propositions, and (3) immediate validation feedback in the form of reformatted raw input. The system's natural syntax and large 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 upon work partially supported by the National Science Foundation under Grant No. 0820138 and Grant No. 0720604.

[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 to a wider audience the benefits of adopting a machine-readable representation of formal arguments. 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 [30].

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 [29,22,1]. 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 or 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 as input arguments written in this manner, 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 [4] echo this in positing a need for "naturality" in a system's concrete representation. The system must provide 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 [4] 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 [17], or logical definitions and theorems [6]) is a natural starting point, real-time keyword-based lookup techniques for programming environments [7,14,15] 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. It can also include notifications 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 [27] or logical system the user wishes to employ), and the speed with which the feedback can be provided (which will often depend on

the validation technique the user chooses).

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. Figure 2 illustrates the user interface from the user's perspective. The user submits a formal *argument* represented using concrete syntax. As the user types, real-time hints for syntax are provided at the top of the interface based on the text surrounding the cursor. The user can 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).

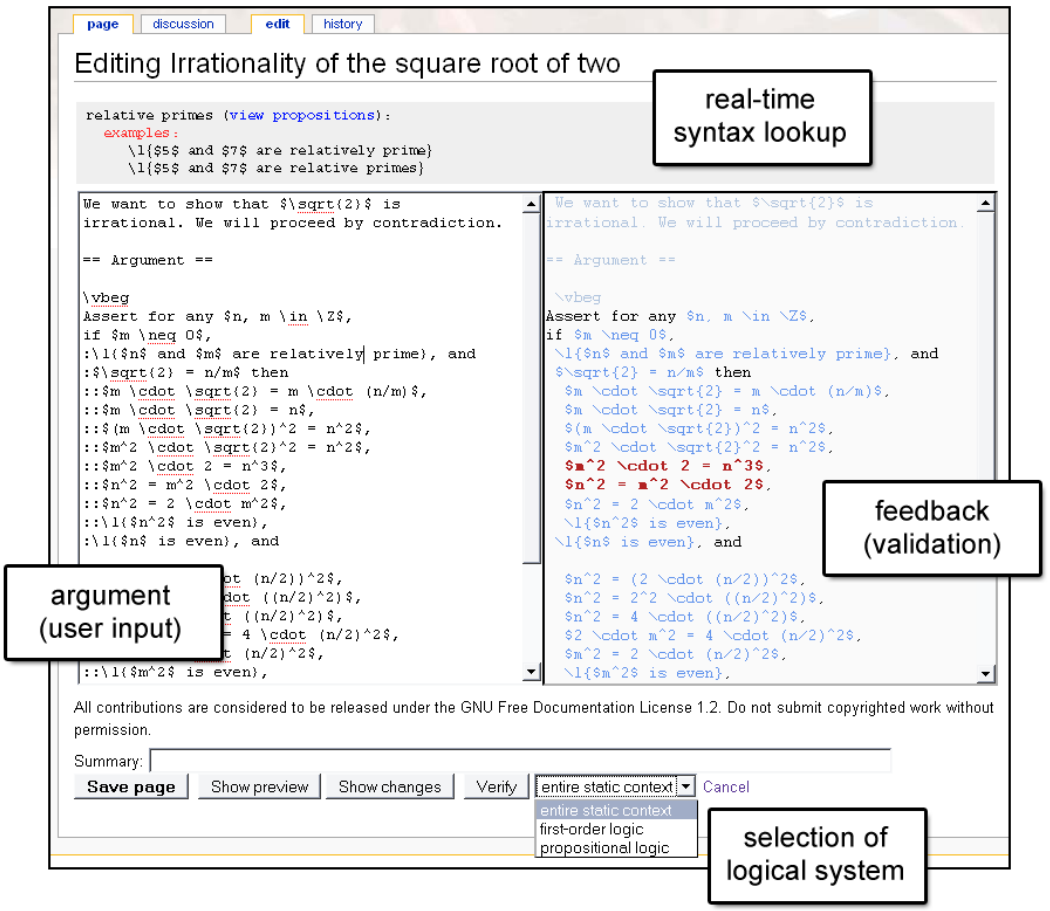

Fig. 2. Screen capture of the user interface.

Figure 3 illustrates the overall organization of the various components of the system, and how they behave in practice. An expert-managed database contains a large 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 machine. The server sends the JavaScript application to the client when the web interface illustrated in Figure 2 first loads. Users then author arguments on their own browser with the help of the JavaScript application, and have the option at any time of submitting their arguments with the click of a button to the server for validation.

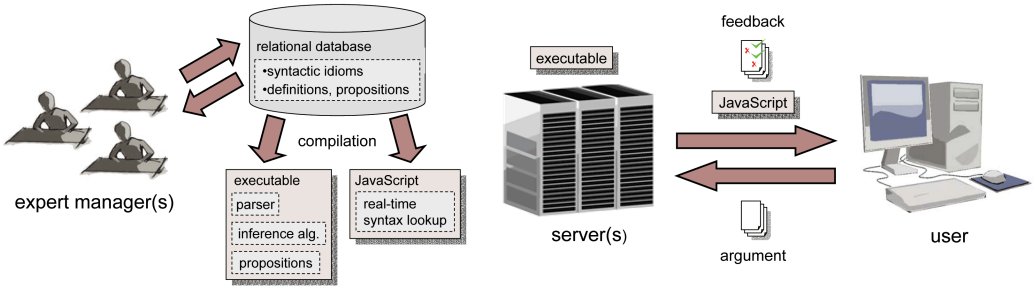

Fig. 3. Overview of system components and operation.

3.1 Concrete Syntax for Arguments

The concrete syntax for arguments (listed in part in Figure 4) 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.

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 [10]) 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 `\p{$p$ is a path in $G$}` is represented using the list

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

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`
		|	$\{e\}$
		|	$e_1, \ldots, e_n$
		|	$e_1 + e_2$
		|	$\vdots$

Fig. 4. Concrete syntax overview.

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 large collection of constants and operators (which is consistent with the basic commands found in existing LaTeX packages where possible) is supported.

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 was constructed in Haskell using the Parsec parser combinator library [13], 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 [23].

3.2 Library Access

The AARTIFACT library of supported propositions and definitions contains a large 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".

It is unreasonable to expect a user to learn all the possible syntactic constructs and idioms for common concepts. Thus, a real-time keyword lookup system is integrated into the user interface. 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 such a scenario. The user's cursor is positioned over text that includes the words "relatively prime". At the top of the interface, a suggestion appears that provides examples of relevant predicates, and a link to view propositions within the library that include this predicate.

In this way, 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 $\setminus$ or $-$, and the notation for concatenation of vectors might be $\cdot$ or $\circ$. 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 2, 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 [12] the AARTIFACT system in defining and reasoning about a compositional formalism for a typed domain-specific language [3]. 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: an advanced undergraduate course on functional pro-

gramming [11] and an introductory undergraduate course in linear algebra. These experiences demonstrated 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. Precise error messages and the ability to select the validation technique are especially important in classroom instruction, because students are new to a system and it is necessary to provide for them a validation procedure that incorporates only a small library of results pertaining to the topic at hand.

5 Related and Future Work

The aartifact syntax reflects the design principles of other formal verification systems such as `Tutch` [1] and Scunak [4]. 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 [4], the ForTheL language and SAD proof assistant [27], the EPGY Theorem-Proving Environment [16], the ΩMEGA proof verifier [25], and in the work of Sieg and Cittadini [24]. 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 [23]. More widely, there exist other efforts to create interfaces and systems for practical formalization of mathematics. The MathLang project [8] 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 [5].

There exists a variety of other tools for formal representation and machine verification of proofs, and many of these have been surveyed and compared along a variety of dimensions [30]. Some of these tools provide a way to construct proofs by induction, such as Coq [19], PVS [18], and Isabelle [20,21]. More specifically, formal representation and verification systems include Isabelle/Isar [28] and Mizar [26]. Our work shares some of the motivations underlying the design of both of these. 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 has been some work in keyword-based lookup involving Mizar [6], but it did not involve providing the user with real-time syntactic and semantic hints.

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 [9], and there exists an online search tool called Hoogle for exploring the Haskell libraries [17]. There has also been work on retrieval of library functions, and even automated construction of programming language code snippets, using collections of keywords [7,14,15]. 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.

References

[1] A. Abel, B. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic. In *IJCAR Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP01)*, 2001.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0. http://research.sun.com/projects/plrg/fortress.pdf, March 2008.

[3] Azer Bestavros, Assaf Kfoury, Andrei Lapets, and Michael Ocean. Safe Compositional Network Sketches: The Formal Framework. In *Proceedings of HSCC'10: The 13th ACM International Conference on Hybrid Systems: Computation and Control (in conjunction with CPSWEEK)*, Stockholm, Sweden, April 2010.

[4] Chad E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *Mathematical Knowledge Management, MKM 2006*, pages 110–123, Wokingham, England, 2006.

[5] Stephen Buswell, Stan Devitt, Angel Diaz, Patrick Ion, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) 1.01 Specification (Abstract). http://www.w3.org/TR/REC-MathML.

[6] Paul Cairns and Jeremy Gow. Integrating searching and authoring in mizar. *J. Autom. Reason.*, 39(2):141–160, 2007.

[7] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input, 2009.

[8] Fairouz Kamareddine and J. B. Wells. Computerizing Mathematical Text with MathLang. *Electron. Notes Theor. Comput. Sci.*, 205:5–30, 2008.

[9] Susumu Katayama. Library for systematic search for expressions. In *AIC'06: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications*, pages 381–387, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[10] Andrei Lapets. Improving the accessibility of lightweight formal verification systems. Technical Report BUCS-TR-2009-015, Computer Science Department, Boston University, April 30 2009.

[11] Andrei Lapets. Lightweight Formal Verification in Classroom Instruction of Reasoning about Functional Code. Technical report, CS Dept., Boston University, November 2009.

[12] Andrei Lapets and Assaf Kfoury. Verification with Natural Contexts: Soundness of Safe Compositional Network Sketches. Technical Report BUCS-TR-2009-030, CS Dept., Boston University, October 1 2009.

[13] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, Departement of Computer Science, Universiteit Utrecht, 2001.

[14] Greg Little and Robert C. Miller. Keyword programming in java, March 2009.

[15] Greg Little, Robert C. Miller, Victoria Chou, Michael Bernstein, Tessa Lau, and Allen Cypher. *Sloppy Programming*. 2009.

[16] David McMath, Marianna Rozenfeld, and Richard Sommer. A Computer Environment for Writing Ordinary Mathematical Proofs. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 507–516, London, UK, 2001. Springer-Verlag.

[17] Neil Mitchell. Hoogle overview. *The Monad.Reader*, (12):27–35, November 2008.

[18] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[19] Catherine Parent-Vigouroux. Verifying programs in the calculus of inductive constructions. *Formal Aspects of Computing*, 9(5-6):484–517, 1997.

[20] L. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.

[21] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and Its Applications*. MIT Press, 1997.

[22] P. Rudnicki. An overview of the Mizar project. In *Proceedings of 1992 Workshop on Types and Proofs for Programs*, 1992.

[23] Sukyoung Ryu. Parsing fortress syntax. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 76–84, New York, NY, USA, 2009. ACM.

[24] Wilfried Sieg and Saverio Cittadini. Normal Natural Deduction Proofs (in Non-classical Logics). In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005.

[25] Jörg H. Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, and Martin Pollet. Proof Development with OMEGA: sqrt(2) Is Irrational. In *LPAR*, pages 367–387, 2002.

[26] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In *Proc. of the 9th IJCAI*, pages 26–28, Los Angeles, CA, 1985.

[27] Konstantin Verchinine, Alexander Lyaletski, Andrei Paskevich, and Anatoly 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*, pages 583–598, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Markus Wenzel and Lawrence C. Paulson. Isabelle/Isar. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 41–49. Springer, 2006.

[29] Markus M. Wenzel. *Isabelle/Isar - A versatile environment for human-readable formal proof documents*. PhD thesis, Institut fr Informatik, Technische Universitt Mnchen, 2002.

[30] Freek Wiedijk. Comparing mathematical provers. In *MKM '03: Proceedings of the Second International Conference on Mathematical Knowledge Management*, pages 188–202, London, UK, 2003. Springer-Verlag.