User-friendly Support for Common Mathematical Concepts in a Lightweight Verifier (Discussion Paper)[*]

Andrei Lapets
Boston University
Boston, USA
lapets@bu.edu

Abstract

Machine verification of formal arguments can only increase our confidence in the correctness of those arguments, but the costs of employing machine verification still outweigh the benefits for some common kinds of formal reasoning activities. As a result, usability is becoming increasingly important in the design of formal verification tools. We describe the AARTIFACT lightweight verification system, designed for processing formal arguments involving basic, ubiquitous mathematical concepts. The system is a prototype for investigating potential techniques for improving the usability of formal verification systems. It leverages techniques drawn both from existing work and from our own efforts. In addition to a parser for a familiar concrete syntax and a mechanism for automated syntax lookup, the system integrates (1) a basic logical inference algorithm, (2) a database of propositions governing common mathematical concepts, and (3) a data structure that computes congruence closures of expressions involving relations found in this database. Together, these components allow the system to better accommodate the expectations of users interested in verifying formal arguments involving algebraic and logical manipulations of numbers, sets, vectors, and related operators and predicates. We demonstrate the reasonable performance of this system on typical formal arguments and briefly discuss how the system's design contributed to its usability in two case studies.

1 Introduction

In research efforts involving mathematical rigor, as well as in mathematical instruction, there exist many benefits to adopting a formal representation that is amenable to machine verification. These benefits 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 exists a variety of such machine verification systems, and some have been surveyed and compared along a variety of dimensions [39].

Unfortunately, the costs of employing machine verification still outweigh the benefits in a variety of formal reasoning activities. While it is by restricting a user to correct arguments that a machine verifier serves its purpose, such restrictions can inhibit even an expert user's productivity when they are reflected in the machine verifier's interface. To date, broad accessibility and quality interface design have not been a priority in the design of machine verification systems. On the contrary, a researcher hoping to enjoy the benefits of formal verification is

[*]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.

presented with a variety of obstacles, both superficial and fundamental. One author, commenting on designing an interface for representing proofs, opines that "we seem to be stuck at the assembly language level" [37]. While employing machine verification systems can improve the accuracy of a user's formal reasoning activities, the systems' counterintuitive or cumbersome interfaces may impair the capacity and productivity of the user.

1.1 Mathematical Reasoning with Common Concepts

```
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.
```

Figure 1: An example of a proof of the infinitude of primes.

Even if one considers a small collection of mathematical concepts, a practicing mathematician is familiar with a large number and a great variety of propositions that describe relationships between the concepts in such a collection. To illustrate this, Figure 1 presents a very short proof of the infinitude of primes. This short 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.

Any system that aims to support the kind of formal reasoning activity users employ in constructing such a proof must have several characteristics. The system must provide a natural syntax that corresponds to the conventions that prevail in the target community of users. The designers of Scunak mathematical assistant [7] echo this in positing a need for "naturality" in a system's concrete representation. It must also provide some basic infrastructure for assembling logical arguments using typical logical constructs (i.e. conjunction, disjunction, quantification). Most importantly, it must not only incorporate an extensive library containing many concepts, properties, and relationships that a user will want to employ, it must allow the user to employ many of these without explicit reference (i.e. it must not *require* the user to name results, or refer to them by name). The designers of the Scunak system [7] 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].

1.2 A User-friendly Tool for Verifying Basic Mathematical Reasoning

The AARTIFACT[1] system is a lightweight verifier for formal arguments [16]. The system provides a familiar concrete syntax for common mathematical concepts that overlaps with English, MediaWiki markup, and LaTeX. The system's flexible parser allows the user to employ a selection of LaTeX constructs for mathematical notation, to use predicates represented as natural language phrases, and to introduce her own constants and infix operators. When a user submits to the system a formal argument for processing, it is the system's responsibility to recognize whether the particular manipulations in the argument are valid based on its ability to refer to a large database of logical propositions involving common mathematical concepts.

2 Design of the Lightweight Verification System

The AARTIFACT verification system allows the user to employ a familiar syntax for formal arguments that is based on LaTeX and English. We discuss the concrete syntax insofar as it is useful to introduce the system's user interface. However, we are primarily interested in discussing three essential underlying components of the system:

(1) a basic logical inference algorithm;

(2) a database of definitions and propositions ("static context") involving common concepts;

(3) a dynamic data structure ("dynamic context") for computing congruence closures of expressions.

In this section we motivate and describe the design of each of these components.

2.1 Syntax, Parser, and Interface

The AARTIFACT system processes formal arguments in the form of ASCII text files. The earlier example presented in Figure 1 can be processed by the AARTIFACT system in the form presented.[2] To present another example that involves more extensive algebraic manipulation, we consider a plain text (interpreted by LaTeX) formal argument that $\sqrt{2}$ is irrational, as seen in Figure 2. The argument is made by assuming the negation of the hypothesis, and concluding with a contradiction.

The AARTIFACT syntax is defined in full detail in an earlier report [16]. The syntax is simple, and is backward- and forward-compatible. That is, it supports only mathematical syntax, and no special syntax that can aid or direct a verification process; it also conforms to many conventions already followed by the target community of users. In adopting such a syntax, this work shares the motivations behind the adoption of similar syntaxes by the designers of the Fortress programming language [2], Scunak mathematical assistant system [7], the ΩMEGA proof verifier [34], and the `Tutch` proof checker [1].

Furthermore, the syntax allows the user to construct her formal argument in any order without any of the restrictions of an interactive proof assistant. As has been observed before [1], an interactive proof assistant that directs the user is actually an inconveniently rigid framework. The user is required to learn how to direct the interactive system, cannot easily "jump around"

[1] Source code (for a Haskell implementation) and a demonstration version of the general-purpose automated assistant, integrated with the MediaWiki online content management system, is available at `http://www.aartifact.org`.

[2] The figure itself contains the output produced by LaTeX, not the ASCII text source.

Assert for any $n, m \in \mathbb{Z}$,
if $m \neq 0$,
n and m are relatively prime, and
$\sqrt{2} = n/m$ then
$m \cdot \sqrt{2} = m \cdot (n/m)$,
$m \cdot \sqrt{2} = n$,
$(m \cdot \sqrt{2})^2 = n^2$,
$m^2 \cdot \sqrt{2}^2 = n^2$,
$m^2 \cdot 2 = n^2$,
$n^2 = m^2 \cdot 2$,
$n^2 = 2 \cdot m^2$,
n^2 is even,
n is even, and
$n^2 = (2 \cdot (n/2))^2$,
$n^2 = 2^2 \cdot ((n/2)^2)$,
$n^2 = 4 \cdot ((n/2)^2)$,
$2 \cdot m^2 = 4 \cdot (n/2)^2$,
$m^2 = 2 \cdot (n/2)^2$,
m^2 is even,
m is even,
$GCF(m, n) \geq 2$,
$GCF(m, n) = 1$, and
there is a contradiction.

Figure 2: Another example of a verifiable formal argument involving algebraic manipulations.

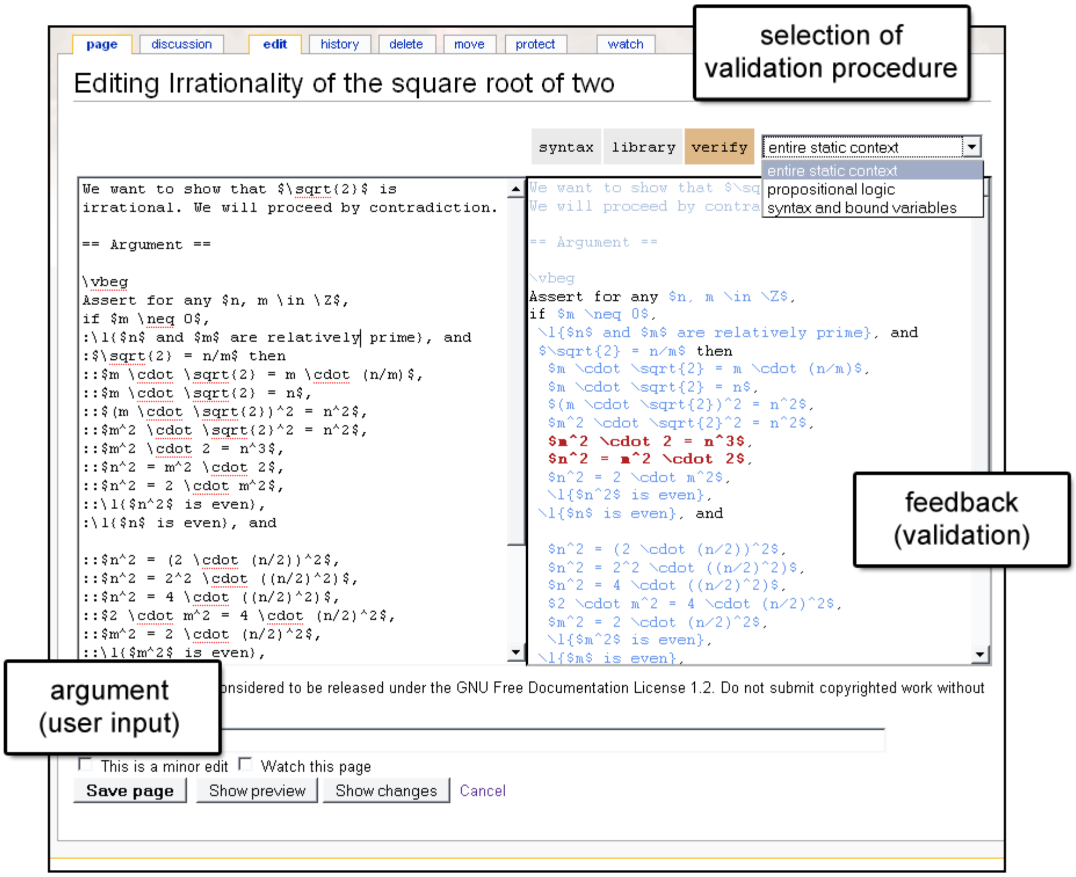
while constructing a proof, and cannot resort to a lightweight approach under which only some parts of the proof are formal and correct. Furthermore, this discourages designers of verification and content management systems from adopting the interface and discourages users from employing the interface as a communication medium.

To illustrate that the chosen concrete representation facilitates integration with a variety of environments and systems, AARTIFACT has been integrated with the MediaWiki content management system.[3] Figure 3 provides a screenshot of a user interacting with this system: the left-hand side of the webpage displays the user's working formal argument, while the right side displays the same argument after it has been processed. In the processed argument, blue text is used for expressions that have been verified, while red text is used for expressions that could not been verified.

2.2 "Lightweight" Logical Inference Rules

The AARTIFACT system's verification capabilities are based on a collection of inference rules corresponding to those found in a typical definition of higher-order logic [15] (i.e. those governing conjunction, disjunction, negation, and quantification), and variants thereof found in common

[3] Available at `http://www.aartifact.org`.



Editing Irrationality of the square root of two

```
We want to show that $\sqrt{2}$ is
irrational. We will proceed by contradiction.

== Argument ==

\vbeg
Assert for any $n, m \in \Z$,
if $m \neq 0$,
:\l{$n$ and $m$ are relatively prime}, and
:$\sqrt{2} = n/m$ then
::$m \cdot \sqrt{2} = m \cdot (n/m)$,
::$m \cdot \sqrt{2} = n$,
::$(m \cdot \sqrt{2})^2 = n^2$,
::$m^2 \cdot \sqrt{2}^2 = n^2$,
::$m^2 \cdot 2 = n^3$,
::$n^2 = m^2 \cdot 2$,
::$n^2 = 2 \cdot m^2$,
::\l{$n^2$ is even},
:\l{$n$ is even}, and

::$n^2 = (2 \cdot (n/2))^2$,
::$n^2 = 2^2 \cdot ((n/2)^2)$,
::$n^2 = 4 \cdot ((n/2)^2)$,
::$2 \cdot m^2 = 4 \cdot (n/2)^2$,
::$m^2 = 2 \cdot (n/2)^2$,
::\l{$m^2$ is even},
```

```
We want to show that $\sq
We will proceed by contra

== Argument ==

\vbeg
Assert for any $n, m \in \Z$,
if $m \neq 0$,
 \l{$n$ and $m$ are relatively prime}, and
 $\sqrt{2} = n/m$ then
  $m \cdot \sqrt{2} = m \cdot (n/m)$,
  $m \cdot \sqrt{2} = n$,
  $(m \cdot \sqrt{2})^2 = n^2$,
  $m^2 \cdot \sqrt{2}^2 = n^2$,
  $m^2 \cdot 2 = n^3$,
  $n^2 = m^2 \cdot 2$,
  $n^2 = 2 \cdot m^2$,
  \l{$n^2$ is even},
 \l{$n$ is even}, and

  $n^2 = (2 \cdot (n/2))^2$,
  $n^2 = 2^2 \cdot ((n/2)^2)$,
  $n^2 = 4 \cdot ((n/2)^2)$,
  $2 \cdot m^2 = 4 \cdot (n/2)^2$,
  $m^2 = 2 \cdot (n/2)^2$,
  \l{$m^2$ is even},
 \l{$m$ is even},
```

onsidered to be released under the GNU Free Documentation License 1.2. Do not submit copyrighted work without

Figure 3: A screen capture of the web interface. The equations highlighted in red in the panel on the right represent unverifiable expressions. They are highlighted because an error has been introduced into the proof (the equation `m^2 \cdot 2 = n ^3` should be `m^2 \cdot 2 = n ^2`).

sequent calculus formulations [10]. The logical inference rules of the system and its validation procedure are presented in more detail in a relevant report [16]. In this section, we briefly discuss the manner in which the rules are used within the system, as well as the motivation behind this approach.

The system relies on a small collection of inference rules of the typical form; for example, the inference rule governing the introduction of conjunction is:

$$[\wedge\text{-Intro}]\ \frac{\Delta;\Phi \vdash e_1 \qquad \Delta;\Phi \vdash e_2}{\Delta;\Phi \vdash e_1 \wedge e_2}.$$

In each rule, Δ represents the set of bound variables and Φ represents an assumption context, the definition and implementation of which is discussed in Section 2.4 below. The validation procedure uses some specified subset of this collection of rules (as determined by the logical system the user wishes to employ) and recursively traverses an argument, expanding Δ and Φ as necessary. It is similar to validation procedures in related work, such as the proof checking algorithm of the Tutch system [1]. The distinguishing feature of our procedure is the more sophisticated assumption context, described in Section 2.4.

To consider an example, the inference rule for the introduction of implication is

$$[\Rightarrow\text{-}\textsc{Intro}]\ \frac{\Delta; \mathsf{closure}(\Phi \cup \{e_1\}) \vdash e_2}{\Delta; \Phi \vdash e_1 \Rightarrow e_2}.$$

When the recursive validation procedure encounters an expression of the form "e_1 implies e_2" within some context represented by Δ and Φ, it first checks whether "e_1 implies e_2" is itself contained within the context Φ (a simple lookup in the dynamic context data structure), and also checks whether a single application of an inference rule to some collection of entries in Φ can be used to derive this expression. If either of these checks succeeds, then this subexpression is valid. If neither apply, the procedure adds e_1 to Φ, computes its closure as described in Section 2.4, and applies itself recursively to e_2.

The verification capabilities provided by the AARTIFACT system are "lightweight" in that no guarantee is made about the logical completeness of the validation process. That is, if the system is unable to verify an assertion, the user can only be certain that the system was unable to find a derivation (even with access to the database of propositions described below in Section 2.3). The user cannot be certain that the assertion is false. Furthermore, only a relative guarantee of consistency is provided. That is, if the system verifies an assertion as true, this only means that a derivation for the assertion exists, and that this derivation uses the assumptions supplied by the user within the argument together with zero or more propositions drawn from the database of propositions described below in Section 2.3. Neither the user's assumptions nor the database of propositions are necessarily restricted to those that are consistent with a particular logic. However, the system's verification capabilities can be made sound with respect to a specific logic by restricting what portion of the database and which inference rules are used by the validation procedure. The ability of the system to ensure relative consistency with respect to a particular logic has been demonstrated for propositional logic and first-order logic.

While it is often desirable to establish the consistency of the inference rules of a system (for example, to accommodate users looking for a strict verification of their arguments with respect to a particular logical system), some users may value usability and flexibility over such consistency. The AARTIFACT system can provide less strict validation capabilities, such as detection of unbound variables and incorrect symbolic manipulations. Users who do not wish, or do not have the expertise, to select a particular logical system for their arguments can still benefit from these features. Furthermore, extending the system is then much easier because it is only necessary to add new propositions to the database described below in Section 2.3. It is also easier to add propositions involving abstract, high-level concepts that cannot (without a great investment of effort) be expressed in a sound manner within a particular logical system.

Such an approach is not without precedent. Recent work on verification systems offers a notion of correctness called "ontological correctness" [36] that deals only with syntax, bound variables, and appropriate use of functions. Alloy [12] provides a finite state space search capability that may not always find a counterexample to an assertion where one might exist, which makes its verification capabilities necessarily inconsistent for some models.

2.3 Static Context: Database of Propositions

The static context is a relational database of simple definitions and propositions involving common mathematical concepts such as numbers, sets, relations, maps, graphs, and so forth.

All of these propositions are of the form

$$\forall \overline{x}, r_1(\overline{u}_1) \wedge \ldots \wedge r_n(\overline{u}_n) \Rightarrow r_{n+1}(\overline{u}_{n+1})$$

where $\overline{x}$ represents a list of variables, $r_1, \ldots, r_{n+1}$ are common relations (such as $\leq$ as well as English predicate such as "X `is a set`"), and the entries $\overline{u}_1, \ldots, \overline{u}_{n+1}$ are either constants or variables drawn from the list $\overline{x}$.

There exists a simple web form that allows an expert designer (or a group of such designers) to submit new formulas or manage the existing formulas within the database. It also allows users to browse and search for formulas by the constants, operators, and predicates they contain. While a formula is implicitly associated with particular disciplines in part by the constants, operators, and predicates that it contains, the database includes a simplistic tagging mechanism to better accommodate categorization of formulas. In particular, it is possible to label a formula with the logical system(s) with respect to which the formula is sound. While the consistency of the overall database is never checked or maintained, this capability allows portions of the database that are consistent with a particular logic to be assembled and to be employed exclusively by the validation procedure.

This database is converted into a component of the AARTIFACT system that is utilized by the dynamic context, described below.

2.4 Dynamic Context: Data Structure of Common Relations

The dynamic context is a data structure that can keep track of all the relevant expressions within an individual argument, as well as all the relationships between the expressions as implied by the static context. It is represented as a hypergraph in which the nodes correspond to equivalence classes of argument expressions, and in which the hyperedges represent common mathematical relations, both low-level (e.g. "$\leq$") and high-level (e.g. "`is a perfect matching corresponding to a bijective map`").

The dynamic context is a data structure that can be represented as a tuple $\Phi = (E, Q, R)$:

E ... the set of all expressions and subexpressions found in a formal argument up to a certain point

Q ... a set of equivalence classes over E

R ... a hypergraph of labelled relations over the set of nodes Q

Notice that the set E should be relatively large for any given argument (but bounded by $O(n^2)$ where n is the length of the argument). If the expression "$x + y > z$" is encountered within a formal argument, then

$$\{x,\ y,\ x + y,\ z,\ >\} \subset E.$$

The set E can contain terms, atoms, and formulas. It is also worth noting that if both "$1 + 2$" and "$2 + 1$" are encountered within an argument, these are stored separately.[4]

The operation used by the validation procedure for introducing a new logical expression e into the dynamic context Φ is denoted by $\Phi \cup \{e\}$. This operation extends E and Q with all the subexpressions found in e. It may also extend R, depending on the form of e. For example, if e is of the form "$1 < 2$", the set E is extended to include the expressions "1", "2", and "$1 < 2$" if it does not already contain them, and R is extended with the entry $(<, 1, 2)$. The data structure

[4] Note that if appropriate propositions representing the commutativity of "+" are found in the static context, the corresponding equivalence classes in Q of "$1+2$" and "$2+1$" will be merged once the closure of the dynamic context is computed.

is also accompanied by a closure operation $\mathsf{closure}(\Phi)$ that computes the closure of an entire hypergraph Φ with respect to the propositions found in the static context. For example, if we have the dynamic context

$$\begin{array}{rcl} E & = & \{1,2,3\} \\ Q & = & \{1,2,3\} \\ R & = & \{(<,1,2),(<,2,3)\} \end{array}$$

and the static context contains the proposition

$$\text{for all } x,y,z. \quad x < y \ \wedge \ y < z \quad \Rightarrow \quad x < z,$$

then the closure operation would add $(<,1,3)$ to the dynamic context. The set of equivalence classes Q is updated whenever two equivalence classes must be merged. This can occur in two ways. It can occur if an expression of the form "$e_1 = e_2$" is introduced into the context by the validation procedure. In this case, if e_1 and e_2 were in separate equivalence classes q_1 and q_2 in Φ, $\mathsf{closure}(\Phi \cup \{e_1 = e_2\})$ produces a new dynamic context in which q_1 and q_2 have been merged. It can also occur if the static context contains a proposition whose conclusion contains an equivalence; for example, consider

$$\text{for all } x,y. \quad x \leq y \ \wedge \ y \leq x \quad \Rightarrow \quad x = y.$$

In this case, merely computing $\mathsf{closure}(\Phi)$ can cause equivalence classes to be merged.

The dynamic context and its accompanying closure operation are very similar to the data structures and algorithms found in work on congruence closures [4]. The contribution of this work is to apply this technique to purely symbolic expressions involving constructs that can correspond to highly abstract but still common mathematical concepts (e.g. perfect matching, acyclic graph, etc.) Furthermore, the closure is computed with respect to a *large* collection of both low-level (e.g. "<") and high-level (e.g. "`is a perfect matching corresponding to a bijective map`") relations.

As mentioned in Section 2.2, the closure computation is integrated into the validation procedure corresponding to the logical inference rules. This means that the dynamic context is extended (and the closure computation is performed) once for every subexpression encountered by the validation procedure as it recursively processes an expression. Consequently, the closure operation must be efficient, and the growth rate of the dynamic context hypergraph must be manageable. The latter issue is discussed in Section 3.1. A full description of the definitions and implementations of the dynamic context and the associated closure algorithm can be found in an earlier report [18].

As a note on usability, it is worth mentioning that by placing a special marker expression anywhere within an argument, the user can retrieve a list of expressions that represents the contents of the hypergraph as it exists when the validation procedure reaches the marker expression. A client-side JavaScript application allows the user to filter this list interactively by using keywords, making more manageable the process of examining the contents of the hypergraph. However, the hypergraph can grow large, which can make impractical its delivery from the verification server over the network back to the user. Further work is required to improve the feasibility of this interactive feature.

3 Performance and General Evaluation

3.1 Growth of the Dynamic Context

Because the dynamic context assembles a hypergraph that can have as a node every single subexpression found in a formal argument, it is natural to consider the growth rate of this data structure during the validation of an argument. We have found that for the formal arguments we have encountered so far in practice, the dynamic context's growth rate is linear. The graph in Figure 4 is produced by measuring the sizes of the components of the dynamic context as the system processes the formal argument presented in Figure 1. Figures 5, 6, and 7 present such graphs for three more examples. Figure 6 corresponds to a linear algebra homework assignment completed by students (as part of the deployment discussed further in Section 3.3 below). Figure 7 represents a very large formal argument: the proof of soundness of the NetSketch formalism [6, 19]. The sizes of the components shrink at certain points because premises can fall out of scope during the validation process. For example, consider the following sequence of statements:

"Assert if $x > y$ then $y < x$. Assert $1 = 1$."

The premise "$x > y$" no longer applies once the first "Assert" statement is verified.

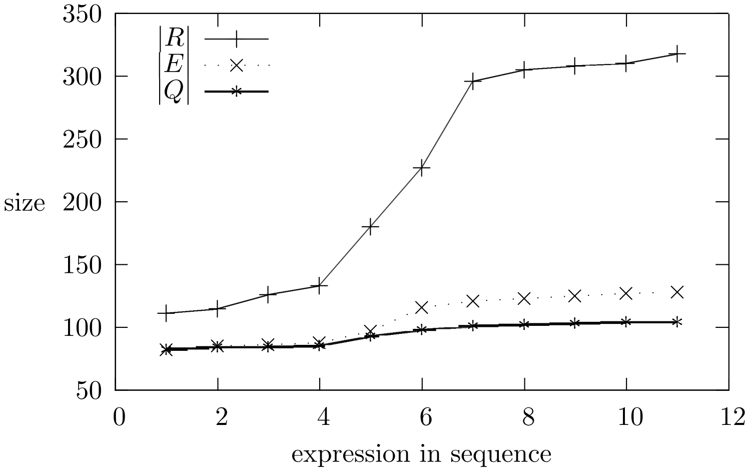


Figure 4: the number of primes is infinite

The linear growth rate in this data suggests that the approach is feasible for many practical applications of a size and complexity that is commensurate with that of the examples we have considered. Further work is needed to identify and mitigate potential worst-case scenarios, however. For example, a sequence of assumptions of the form

$$a_1 < a_2, \quad a_2 < a_3, \quad \ldots, \quad a_{39} < a_{40}.$$

leads to quadratic growth in the size of the dynamic context components, as illustrated in Figure 8.

3.2 Usability in Research Applications

We have utilized [19] the AARTIFACT system in defining and reason about a novel compositional formalism underlying a typed domain-specific language [6]. This formalism can be used to model

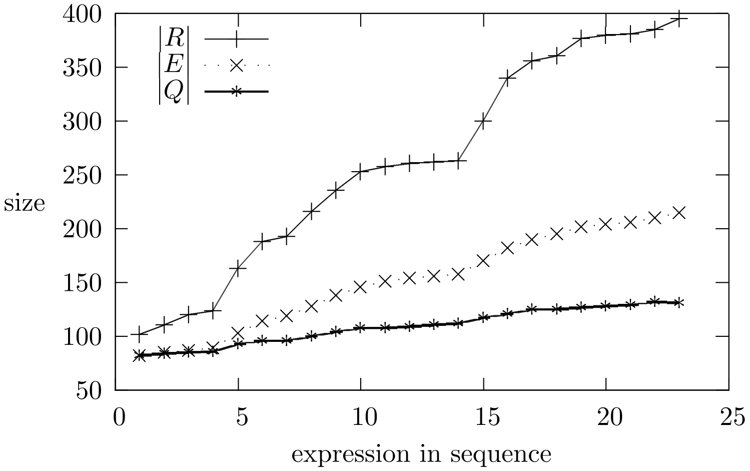


Figure 5: $\sqrt{2}$ is irrational

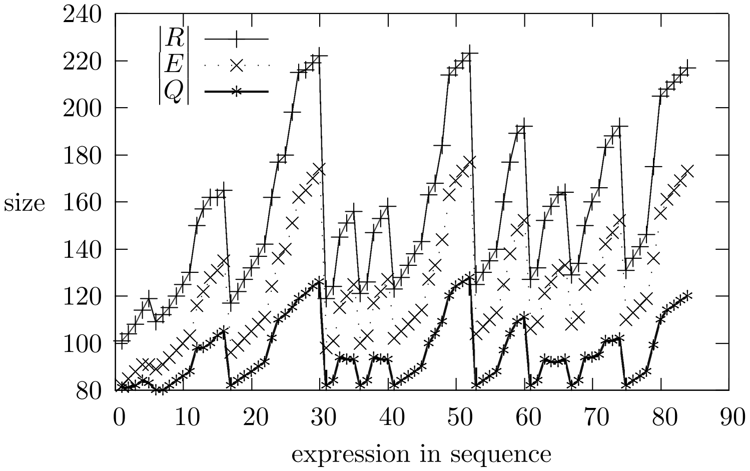


Figure 6: $\mathbb{R}$ is a vector space $\Rightarrow$ $\mathbb{R}^2$ is a vector space

and assemble networks, and to reason about and analyze constraints on flows through these networks. We assembled a verifiable proof that this formalism is sound with respect to its semantics.

This exercise demonstrated some of the advantages of the design of the AARTIFACT system. We were able to define the semantics for our formalism in a machine-readable representation that is also highly accessible to humans. We were able to use LaTeX syntax and to introduce user-defined infix operators thanks to the support provided by the flexible interface and parser. We were also able to take full advantage of the system's support for reasoning about concepts in set theory (provided by a large collection of propositions in the database that deal with sets and related concepts) by employing, without explicit references, laws that govern the relationships between common operations on sets.

Despite the fact that only lightweight verification was employed, the formal assembly process led to the discovery of a few minor errors, and to the simplification of a few side conditions

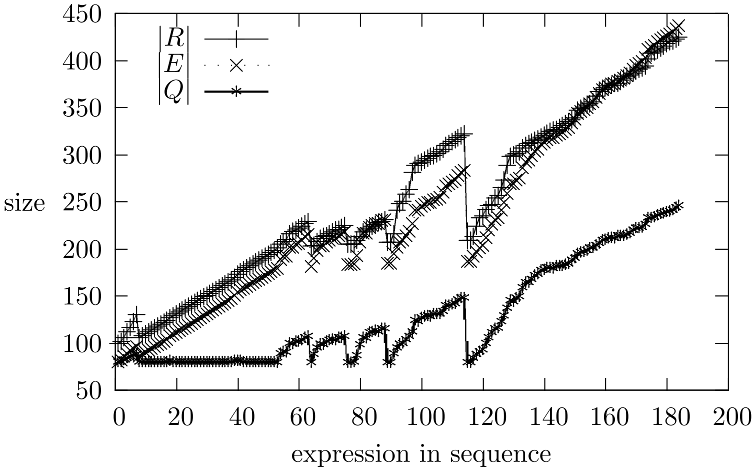


Figure 7: soundness of the NetSketch formalism

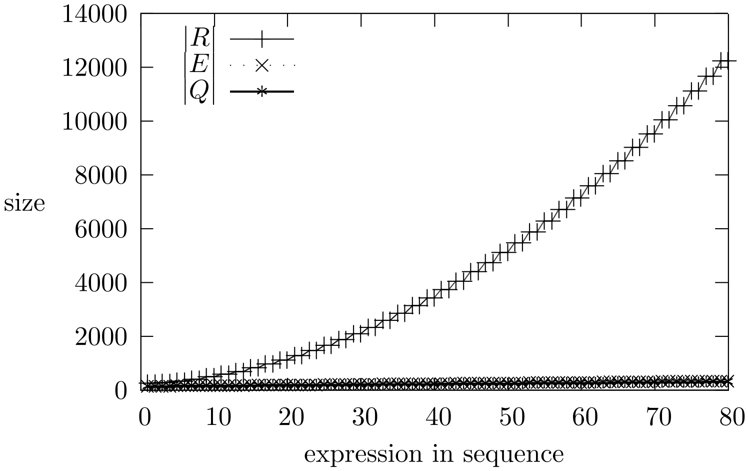


Figure 8: complete graph for "<"

and definitions. The lightweight approach was actually beneficial in allowing us to easily move around verified chunks of an argument without concern for context, something that would be difficult to do when using an interactive theorem proving environment. The lightweight approach also allowed us to introduce and utilize a few lemmas without an explicit proof.

3.3 Usability in Classroom Instruction

The AARTIFACT system has been deployed within two undergraduate courses: an advanced undergraduate course on functional programming [17], and an introductory undergraduate course in linear algebra.[5] These deployments served as a means both for evaluating the usability of

[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.

the AARTIFACT system and for identifying necessary improvements to the system's capabilities.

We observed a few pedagogical advantages to using our formal reasoning system within classroom instruction. The presence of an application that provides direct feedback about formal arguments makes it possible to easily present and reinforce a precise list of valid formal manipulations. The use of a parser combinator library in the implementation of the formal reasoning system's parser (as discussed briefly in previous work [16]) makes it possible to support multiple parsing regimes, including one that corresponds almost exactly to the syntax of the particular programming language in which the formally analyzed code is written. Admittedly, the Parsec parser combinator library we employed [21] was designed with a functional language in mind, but we believe its flexibility would make it similarly easy to adjust the system for formal reasoning exercises involving other functional programming languages.

One frequent concern about any sort of automated search or inference done by a formal reasoning system is that the user will become frustrated because she cannot predict what the system might be inferring, or what the system's limitations are. However, within this deployment we have found that no such frustration occurs when the system's capabilities are intuitive and can be described in a succinct and straightforward manner. The syntax of the AARTIFACT system did not present much difficulty to the students in either course. For the functional programming course, the mathematical syntax of AARTIFACT was augmented with typical Haskell operators and looked very similar to Haskell syntax. For the course on linear algebra, the syntax used was a subset of LaTeX and was natural enough that many students were able to utilize by consulting nothing more than an example of an argument. As can be seen in more detailed data from the linear algebra course, presented in a related report [20], at least 10 of 16 students were able to complete at least 80% of the required automatically verifiable proofs under these conditions. It is also worth noting that some students used the AARTIFACT syntax in writing their pencil and paper exam solutions (without access to the verifier). It is debatable what this might indicate about their understanding of the formal reasoning techniques they employ, but it does demonstrate that the syntax is not too cumbersome to be used manually (most likely because it corresponds closely to the syntax humans naturally use).

The implicit manipulations that could be verified thanks to the dynamic context were understood by many students without any explicit guidance beyond the presentation of an example. This indicates that the semantics of the system corresponded well to the existing expectations of students who had already been introduced to the mathematical conventions governing the concepts involved.

4 Related Work

The accessible interface of the automated assistant utilized in this work reflects the design principles of other formal verification systems such as `Tutch` [1] and Scunak [7]. The need for natural interfaces (both superficial and functional) in automated verification has been recognized to varying degrees by the designers of the `Tutch` proof checker [1], the Scunak mathematical assistant system [7], the ForTheL language and SAD proof assistant [36], the EPGY Theorem-Proving Environment [23], the ΩMEGA proof verifier [34], the ProveEasy system [8], in the work of Sieg and Cittadini [33], and in the work of Hallgren and Ranta [11]. The ontology-oriented, lightweight verification capabilities of the automated assistant are inspired by work in the assembly of large-scale formal and semi-formal ontologies [29, 22].

Relevant work on retrieval and application of propositions by structure has been done within the context of Haskell in the development of search tools that allow users to retrieve and browse expressions within a context by their type [14], and there exists an online search tool called

Hoogle for exploring the Haskell libraries [24]. Matita [3] is a proof assistant the automation of which is heavily based on an integrated search engine. Developing a robust and extensive construct with these kinds of capabilities within the context of formal reasoning is essential, and could even lead to support for reasoning by analogy. As observed by others working in this area [1], this would be beyond the current state of the art.

Our notion of a dynamic context is a variant of a *congruence closure* [4]. A congruence closure can be used to implement a context-directed inference algorithm for finite collections of concepts or expressions introduced by the user. This is achieved by considering whether logical expressions are equivalent to the constant term representing "true". Work exists on the efficiency of algorithms for computing congruence closures [25, 26]. Related work in the construction of SMT solvers [27], and especially general-purpose, multi-domain SMT solvers [5, 9] is also relevant. Such systems integrate multiple algorithms and techniques within a single tool. This allows them to provide some verification and computation capabilities for formulas that involve predicates and operators from undecidable theories.

More widely, there exist other efforts to create interfaces and systems for practical formalization of mathematics. The MathLang project [13] is an extensive, long-term effort that aims to make natural language an input method for mathematical arguments and proofs. There also exist a variety of tools for formal representation and machine verification of proofs, and many of these have been surveyed and compared along a variety of dimensions [39]. Some of these tools provide a way to construct proofs by induction, such as Coq [30], PVS [28], and Isabelle [31, 32]. However, these systems usually require users to consult documentation and to have some understanding of logic and formal systems before they 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. Our work shares some of the motivations underlying the design of both Isabelle/Isar [38] and Mizar [35]. 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.

5 Conclusion and Future Work

We have described the AARTIFACT lightweight verification system, as well as some of the motivation behind its design. A verification system can provide a familiar, friendly syntax that is independent of the strategies used by the underlying verifier, and a lightweight verifier can be augmented with a data structure for computing congruence closures to further enhance its usability. We have demonstrated that such a system can have reasonable performance on real-world examples of formal arguments and discussed the implications of the system's characteristics in actual applications.

Further extensions to the static and dynamic contexts are possible. In particular, it should be possible to extend support to slightly more complex propositions, particularly those involving at least one existential quantifier, or even those containing higher-order predicates. Ensuring that the defined algorithms converge under such a scheme would make for an interesting challenge. It is also necessary to better characterize "typical" formal arguments, and perhaps even to detect unusual arguments that can cause the dynamic context to grow to an unmanageable size. Better facilities for interacting with a dynamic context in real time (without waiting for the validation procedure to process an argument) would greatly enhance the user experience.

References

[1] A. Abel, 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] 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. March 2008.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reason.*, 39(2):109–139, 2007.

[4] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 64–78, London, UK, 2000. Springer-Verlag.

[5] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 924–929, San Jose, CA, USA, 2007. EDA Consortium.

[6] Azer Bestavros, Assaf Kfoury, Andrei Lapets, and Michael 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)*, pages 231–241, Stockholm, Sweden, April 2010.

[7] Chad E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *MKM '06: Mathematical Knowledge Management*, pages 110–123, Wokingham, England, 2006.

[8] Rod M. Burstall. Proveeasy: helping people learn to do proofs. *Electr. Notes Theor. Comput. Sci.*, 31:16–32, 2000.

[9] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.

[10] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Co., Amsterdam, 1969. Edited by M. E. Szabo.

[11] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *LPAR '00: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, pages 70–84, Berlin, Heidelberg, 2000. Springer-Verlag.

[12] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

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

[14] 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).

[15] Stephen Cole Kleene. *Mathematical Logic.* Dover Publications, 1967.

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

[17] Andrei Lapets. Lightweight Formal Verification in Classroom Instruction of Reasoning about Functional Code. Technical Report BUCS-TR-2009-032, CS Dept., Boston University, November 2009.

[18] Andrei Lapets and David House. Efficient Support for Common Relations in Lightweight Formal Reasoning Systems. Technical Report BUCS-TR-2009-033, CS Dept., Boston University, November 2009.

[19] 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.

[20] Andrei Lapets and Assaf Kfoury. A User-friendly Interface for a Lightweight Verification System. Technical Report BUCS-TR-2010-011, CS Dept., Boston University, May 2010.

[21] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Departement of Computer Science, Universiteit Utrecht, 2001.

[22] H. Liu and P. Singh. Conceptnet — a practical commonsense reasoning tool-kit. *BT Technology Journal*, 22(4):211–226, 2004.

[23] 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.

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

[25] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[26] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[27] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

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

[29] Kathy Panton, Cynthia Matuszek, Douglas Lenat, Dave Schneider, Michael Witbrock, Nick Siegel, and Blake Shepard. Common Sense Reasoning – From Cyc to Intelligent Assistant. In Yang Cai and Julio Abascal, editors, *Ambient Intelligence in Everyday Life*, volume 3864 of *LNAI*, pages 1–31. Springer, 2006.

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

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

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

[33] 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.

[34] 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.

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

[36] 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.

[37] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 167–184, London, UK, 1999. Springer-Verlag.

[38] 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.

[39] 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.