Improving the accessibility of lightweight formal verification systems

Andrei Lapets

Abstract. In research areas involving mathematical rigor, there are numerous benefits to adopting a formal representation of models and arguments: reusability, automatic evaluation of examples, and verification of consistency and correctness. However, broad accessibility has not been a priority in the design of formal verification tools that can provide these benefits. We propose a few design criteria to address these issues: a simple, familiar, and conventional concrete syntax that is independent of any environment, application, or verification strategy, and the possibility of reducing workload and entry costs by employing features selectively. We demonstrate the feasibility of satisfying such criteria by presenting our own formal representation and verification system. Our system's concrete syntax overlaps with English, LaTeX and MediaWiki markup wherever possible, and its verifier relies on heuristic search techniques that make the formal authoring process more manageable and consistent with prevailing practices. We employ techniques and algorithms that ensure a simple, uniform, and flexible definition and design for the system, so that it easy to augment, extend, and improve.

Mathematics Subject Classification (2000). Primary 68Q60; Secondary 65G20, 68N17.

Keywords. Specification, verification.

1. Introduction

In research areas involving mathematical rigor, as well as in mathematical instruction, there exist many benefits to adopting a formal representation. These include reusability, automatic evaluation of examples, and particularly the opportunity to employ formal verification systems. Such systems can offer anything from detection of basic errors, such as unbound variables and type mismatches, to full

.

confidence in an argument because it is verifiably constructed using the fundamental principles of a consistent logic. However, to date, broad accessibility has not been a priority in the design of formal verification systems that can provide these benefits. On the contrary, a researcher hoping to enjoy the benefits of formal verification is presented with a variety of obstacles and shortcomings, both superficial and fundamental. Consequently, instructors and researchers have so far chosen to ignore such systems; in the literature in most domains of computer science and mathematics there are only isolated attempts to include machine-verified proofs of novel research results, and in only a few mathematics and computer science courses are such systems employed in presenting material or authoring solutions to assignments.

1.1. Obstacles

We briefly summarize some potential obstacles and disincentives for using formal representation and verification systems. We believe these problems are likely contributors to the perceived difficulty of constructing and maintaining formal representations of arguments. In turn, these lead many researchers to approach formal representation and verification systems with skepticism. In the following, we imagine the user to be a researcher that intends to begin formalizing some novel, potentially unfinished results, and is considering whether to use a formal representation and verification system with which she is *not yet familiar*.

Unfamiliar syntax and/or environment. In most cases, the user must familiarize herself with a new syntax, or worse, an entirely new editing application or environment. She may even be limited to using only particular applications or environments for her desired verification tool, such as a user of the MathLang system [6]. Furthermore, it is possible that the domain-specific notation she wishes to use is not supported directly by the system, and the system cannot be modified easily to accommodate such a notation. Thus, the researcher may need to laboriously translate existing concepts into a representation that does not retain the familiar conventions prevalent in her own community. Adopting the new notation may also limit the ability of others to understand the formalized argument, impeding communication.

Unnatural, cumbersome bottom-up structure. Most systems allow arguments to be built up from the fundamental principles of particular logics. Such systems include Coq [8], Isabelle/Isar [13], Mizar [12], and PVS [7]. In order to formally represent sophisticated and novel arguments in a domain of research, a wide variety of underlying results and assumptions must be assembled. For researchers who are already familiar with the fundamental results in their domain and who are focused on obtaining novel results, assembling such a library is a daunting and inherently uninteresting task. However, it is exactly such researchers who are most qualified to assemble such a library. Typically, systems are not designed with this process in mind, and no guidance is provided for how best to coordinate or avoid it.

Lack of options in selecting benefits. It is possible that a researcher wishes to employ only certain benefits of a formal representation, such as unbound variable

detection, and does not wish to invest any extra effort to enjoy additional benefits. For example, it may be the case that the researcher would not have full confidence in arguments if they were based on the fundamental principles of a logic because they are not certain the logic is an adequate model for their purposes. This is a common concern in research on the safety of cryptographic protocols [1]. However, most verification tools either do not allow such a selection to be made, requiring that all proofs be constructed from fundamental principles of logics (as in Coq, Isabelle/Isar, Mizar, and PVS), or limit the user once a commitment has been made to using only the benefits provided by the chosen system, with no possibility of introducing a more rigorous approach in the future using a *compatible* representation (as in the Alloy modelling language [5]).

1.2. Our Approach

To address these issues, we advocate a formal representation and verification system design that couples a familiar and simple concrete representation with a verification approach that facilitates forward compatibility and gradual evolution of both the verifier and the knowledge base. We believe that the viability of our approach is demonstrated by the formal representation and verification system presented in this work. We summarize our main design principles and briefly discuss how they manifest themselves in our system.

Familiar, simple concrete syntax. We adopt a familiar concrete syntax that overlaps with English, MediaWiki markup, and LaTeX wherever possible. The user may use a selection of LaTeX constructs in mathematical notation, and can use English phrases as predicates. The syntax also allows the user to succinctly specify which part of a document is written in a formal manner. This approach ensures *backward compatibility*, in that it is possible to leverage existing knowledge: a user familiar only with LaTeX and English only needs to learn three or four simple syntactic constructs and conventions to begin writing formal arguments immediately. Because the representation itself is no more complex than LaTeX (and actually much simpler), it is not necessary to use any special editor, unless one is desired. Naturally, designers can also construct applications and editors that target this representation, and any editing tool can easily be integrated with our system. We have demonstrated this flexibility by integrating it with the MediaWiki content management system.[1]

Implicit arguments, explicit results. The logical and mathematical concrete syntax in our system only allows the user to represent static expressions, and no syntactic constructions are provided for "helping" a particular verification strategy (e.g. by specifying which axiom or theorem is being applied at a particular point in a formal argument). We believe that this is a much more natural representation for formal arguments: most of the rules being applied in a typical proof found in the literature are *not explicitly mentioned*. Such a representation also ensures that the representation is *forward compatible*: special annotations that might

[1] http://www.safre.org

not be necessary in the future are left out of the formal representation, so that future systems need not deal with parsing special annotations that are no longer necessary. A gradually evolving variety of proof search techniques, tactics, and heuristics can then be included in verification tools that operate on this representation, as the representation level makes no explicit mention of them. It is worth noting that nothing prevents any particular verifier from treating idiomatic expression patterns in a special manner, and we advocate such a design. We use this very approach in dealing with the verification of assertions involving existential quantifiers, described in Section 4.

These two design principles facilitate a formal reasoning process that is more manageable and gradual. Even in the absence of a supporting library for her domain, the user can immediately begin authoring formal arguments and enjoying the benefits of detection of simple errors, such as unbound variables or basic type mismatches. If it is used in this way, the tool is not unlike modelling tools such as Alloy [5], which can only guarantee relative consistency: provided the assumptions and definitions introduced by the user are consistent with whatever entities the user wishes to model, the derived assertions will also be consistent. If, at a later point in time, the user wishes to introduce additional detail that increases confidence in her arguments, she can limit the underlying inference rules, and construct a rich library of results derived from a particular logic. Thus, the user can use a top-down approach, starting with desired novel results, and formalizing the details when there is an opportunity to do so. Furthermore, the explicit assumptions accumulated throughout the process can be used as a guide in constructing domain-specific libraries of rigorous results. This leads to a natural, need-based process for assembling libraries, and one that does not encumber the user when is focusing on work that is of interest to her.

The rest of the paper is organized as follows. Section 2 describes the concrete syntax designed for our formal representation and verification system, and illustrates how it might be used. Section 3 describes the abstract syntax and logical inference rules. Section 4 describes the basic verification algorithm and how it can be extended with additional heuristic proof search components. We then review related work, including a very similar project involving the ΩMEGA proof verifier [11], in Section 5. Finally, we summarize and mention possible directions for further work in Section 6.

2. Concrete Syntax and Interface

2.1. Example

In Table 1, we present a concrete representation of a simple variant of an argument that $\sqrt{2}$ is irrational, already processed by LaTeX. The actual concrete syntax can be seen in Table 2. It is worth noting that the entire LaTeX document of this article was supplied to our system. The source text corresponding to the example was successfully parsed, and it was verified that the assertions in the

argument are consistent with respect to the explicitly stated assumptions. We believe that this example illustrates effectively our main points. In particular, syntactically, the argument consists of familiar English phrases and mathematical notation. Semantically, despite the fact that the argument stands alone, *some* verification has already taken place, increasing our confidence in the argument's validity. As in most proofs written informally, the argument's representation never specifies the result being applied at each subsequent step in the proof and contains no references to a verification system or any particular verification strategies. It is the responsibility of our system's verification algorithm to determine which axioms or results are being applied in each part of the argument. However, nothing prevents the user from supplying additional information indicating how each part of the argument is justified; she can either supply this information as part of the informal text, or she can add explicit formal expressions that correspond to the application of the rules in question.

Note that in terms of organization, this example is a *document*, and consists of a sequence of *statements* (such as "Assume" and "Assert"). Each statement contains logical *expressions*: "Assume" allows the user to state a logical expression without verification, and "Assert" allows the user to verify that an expression is logically derivable from the assumptions and assertions that occur before it. Our system *intentionally* does not provide a rich proof language for constructs such as definitions, lemmas, theorems, and so forth. This reduces the amount of new, specialized syntax and semantics a user must learn when employing our system. These structures can easily be introduced outside of the formal syntax based on the needs and preferences of the user or community of users.

While the system's parser supports a variety of English phrases corresponding to common logical operations (such as implication, conjunction, and quantification), a new user only needs to learn one particular phrase for each operation (amounting to no more than five English phrases) or none at all, if the user prefers to use the familiar logical symbols $\wedge$ $\Rightarrow$, $\forall$, $\exists$, and so on. The user is free to use her own English phrases as predicate constants, and these phrases can contain expression parameters. Our system tries to accommodate a small collection of punctuation symbols commonly used grammatically and in LaTeX and MediaWiki formatting, allowing them to be placed between English phrases and mathematical expressions.

We believe that our chosen concrete representation facilitates integration with a variety of environments and systems. For example, by integrating our verifier with the MediaWiki content management system,[2] it is possible to assemble a library of interdependent results simply by adding a command for document inclusion. We have successfully performed such an integration as a demonstration, and the application can be accessed online.[3]

[2] www.mediawiki.org
[3] www.safre.org

Table 1. Example of a verifiably consistent argument.

First, we introduce some assumptions about integers and rational numbers. Assume for any $i \in \mathbb{Z}$, there exists $j \in \mathbb{Z}$ s.t. $i = 2 \cdot j$ implies that i is even. Assume for any $i \in \mathbb{Z}$, i^2 is even implies that i is even. Assume for any $i \in \mathbb{Z}$, i is even implies that there exists $j \in \mathbb{Z}$ s.t. $i = 2 \cdot j$. Assume that for any $q \in \mathbb{Q}$, there exist $n \in \mathbb{Z}, m \in \mathbb{Z}$ s.t. n and m have no common factors and $q = n/m$. Assume for any $x, y, z \in \mathbb{R}$, if $z = x/y$ then $y \cdot z = x$. Assume that if there exist $i, j \in \mathbb{Z}$ s.t. i is even, j is even, and i and j have no common factors then we have a contradiction.
Now, we present our main argument. We want to show that $\sqrt{2}$ is irrational. We will proceed by contradiction. Assume that $\sqrt{2} \in \mathbb{Q}$. Assert that there exist $n, m \in \mathbb{Z}$ s.t. n and m have no common factors, $\sqrt{2} = n/m$. Therefore, $m \cdot \sqrt{2} = n$, $(m \cdot \sqrt{2})^2 = n^2$, $m^2 \cdot \sqrt{2}^2 = n^2$, $m^2 \cdot 2 = n^2$, $n^2 = m^2 \cdot 2$, $n^2 = 2 \cdot m^2$, and so, n^2 is even. Thus, n is even. Furthermore, there exists $j \in \mathbb{Z}$ s.t. $n = 2 \cdot j$, $n^2 = (2 \cdot j)^2$, $n^2 = 2^2 \cdot j^2$, $n^2 = 4 \cdot j^2$, $2 \cdot m^2 = 4 \cdot j^2$, $m^2 = 2 \cdot j^2$, m^2 is even and m is even. Thus, we have a contradiction.

2.2. Parsing

The parser for the concrete syntax was constructed in Haskell using the Parsec parser combinator library [2], which is expressive enough for constructing infinite-lookahead parsers for general context-sensitive grammars. We found the library was simple to use, allowed for a succinct parser implementation, and resulted in a parser that performed without noticeable delay on all the inputs we have produced (we used the infinite lookahead capability at only a few points in our parser definition). Error messages for syntax errors are not ideal, but the Parsec library does provide facilities for producing error messages that specify the location of a parsing error, as well as additional information the designer of the language may wish to specify. We do not believe that we have exhausted the Parsec library's full potential for generating useful error messages in our current implementation.

TABLE 2. Example of a verifiably consistent argument.

```
We introduce some assumptions about integers and rational numbers.

\vbeg
Assume for any $i \in \Z$, {there exists $j \in \Z$ s.t. $i
= 2 \cdot j$} implies that \p{$i$ is even}.

Assume for any $i \in \Z$, \p{$i^2$ is even} implies that \p{$i$
is even}.

Assume for any $i \in \Z$, \p{$i$ is even} implies that there
exists $j \in \Z$ s.t. $i = 2 \cdot j$.

Assume that for any $q \in \Q$, there exist $n \in \Z, m \in \Z$
s.t. \p{$n$ and $m$ have no common factors} and $q = n/m$.

Assume for any $x,y,z \in \R$, if $z = x/y$ then $y \cdot z = x$.

Assume that if there exist $i, j \in \Z$ s.t. \p{$i$ is even},
\p{$j$ is even}, and \p{$i$ and $j$ have no common
factors} then \p{we have a contradiction}.
\vend

Now, we present our main argument. We want to show that $\sqrt{2}$
is irrational. We will proceed by contradiction.

\vbeg
Assume that $\sqrt{2} \in \Q$. Assert that there exist $n, m
\in \Z$ s.t. \p{$n$ and $m$ have no common factors}, $\sqrt{2} =
n/m$.

Therefore, $m \cdot \sqrt{2} = n$,
    $(m \cdot \sqrt{2})^2 = n^2$,
    $m^2 \cdot \sqrt{2}^2 = n^2$,
    $m^2 \cdot 2 = n^2$,
    $n^2 = m^2 \cdot 2$,
    $n^2 = 2 \cdot m^2$,
and so, \p{$n^2$ is even}. Thus, \p{$n$ is even}.

Furthermore, there exists $j \in \Z$ s.t. $n = 2 \cdot j$,
    $n^2 = (2 \cdot j)^2$,
    $n^2 = 2^2 \cdot j^2$,
    $n^2 = 4 \cdot j^2$,
    $2 \cdot m^2 = 4 \cdot j^2$,
    $m^2 = 2 \cdot j^2$,
    \p{$m^2$ is even} and \p{$m$ is even}.
Thus, \p{we have a contradiction}.
\vend
```

Table 3. Abstract syntax.

constants	c	::=	numeric literals, operator symbols, English phrases, ...
expressions	e	::=	$c \mid x \mid e_1 e_2 \mid (e_1, \ldots, e_n)$
		$\mid$	$e_1 \wedge e_2 \mid e_1 \Rightarrow e_2 \mid \forall \overline{x}.e \mid \exists \overline{x}.e$
statements	s	::=	Assume $e \mid$ Assert $e \mid$ Intro $\overline{x}$

3. Abstract Syntax and Logical Inference Rules

We first present the abstract syntax for documents, statements, and expressions, and then define the inference rules that determine what kinds of expressions represent "true" assertions.

3.1. Abstract Syntax

Let X be the set of variables. We denote by x a single variable, and by $\overline{x}$ a vector of variables. We sometimes also denote by $\overline{x}$ the *set* of variables found in $\overline{x}$ (thus, $\overline{x} \subset X$). In Table 3 we summarize the abstract syntax for our representation and verification system. Note that there are only *three* kinds of statements, and two of them ("Assume" and "Assert") are very similar from the user's perspective. The expression syntax corresponds to the typical syntax of any higher-order logic. Note that logical negation and disjunction are not currently supported by the verifier's built-in inference rules and the corresponding search algorithms, but users are free to define and use axioms involving these operators, as these operations are defined as constants.

3.2. Generic Inference Rule Template for Expressions

We define $FV(e)$ to be the collection of free variables in an expression e. The variable Δ will always denote a context of bound variables, and Φ will always denote a set of "true" expressions in a given context. The judgment $\Delta, \Phi \vdash e$ indicates that the expression e has no free variables other than those in Δ, that all expressions in Φ also have no free variables other than those in Δ, and that we consider e to represent a "true" statement in some sense. We define in Table 4 the logical inference rules that define how such judgments on expressions can be constructed.

We call these rules a *template* because they are parameterized by a subset D of the space of syntactically well-formed expressions. This parametrization is useful for ensuring the consistency of the inference rules, as we will briefly mention below in Section 3.3. For a fixed subset D of expressions, we define a space of substitutions $X \to D$. For any vector of variables $\overline{x}$ and any vector of expressions $\overline{e}$ such that each component of $\overline{e}$ is in D, we denote by $[\overline{x} \mapsto \overline{e}]$ the operator that

TABLE 4. Template for logical inference rules for expressions.

$$[\text{ASSUMPTION}]\ \frac{e \in \Phi \quad FV(e) \subset \Delta}{\Delta;\Phi \vdash e}$$

$$[\text{IMPLIES-INTRO}]\ \frac{\Delta;\Phi \cup \{e_1\} \vdash e_2}{\Delta;\Phi \vdash e_1 \Rightarrow e_2} \qquad [\text{IMPLIES-ELIM (MODUS PONENS)}]\ \frac{\Delta;\Phi \vdash e_1 \Rightarrow e_2 \quad \Delta;\Phi \vdash e_1}{\Delta;\Phi \vdash e_2}$$

$$[\wedge\text{-INTRO}]\ \frac{\Delta;\Phi \vdash e_1 \quad \Delta;\Phi \vdash e_2}{\Delta;\Phi \vdash e_1 \wedge e_2} \qquad [\wedge\text{-ELIM-L}]\ \frac{\Delta;\Phi \vdash e_1 \wedge e_2}{\Delta;\Phi \vdash e_1} \qquad [\wedge\text{-ELIM-R}]\ \frac{\Delta;\Phi \vdash e_1 \wedge e_2}{\Delta;\Phi \vdash e_2}$$

$$[\forall\text{-INTRO}]\ \frac{\Delta,\overline{x};\Phi \cap \{e \mid FV(e) \cap \overline{x} = \emptyset\} \vdash e}{\Delta;\Phi \vdash \forall \overline{x}.e} \qquad [\forall\text{-ELIM}]\ \frac{\Delta;\Phi,\vdash \forall \overline{x}.e \quad |\overline{e}| = |\overline{x}| \quad \overline{e} \subset D \quad FV(\overline{e}) \subset \Delta}{\Delta;\Phi \vdash e[\overline{x} \mapsto \overline{e}]}$$

$$[\exists\text{-INTRO}]\ \frac{\Delta;\Phi \vdash e[\overline{x} \mapsto \overline{e}] \quad |\overline{e}| = |\overline{x}| \quad \overline{e} \subset D \quad FV(\overline{e}) \subset \Delta}{\Delta;\Phi,\vdash \exists \overline{x}.e}$$

performs a capture-avoiding substitution in the usual manner on any expression (replacing each x in $\overline{x}$ with the corresponding e' in $\overline{e}$). If $\theta = [\overline{x} \mapsto \overline{e}]$, then $\theta(e)$ and $e[\overline{x} \mapsto \overline{e}]$ both denote the application of this operator on an expression e.

3.3. Consistency of the Inference Rules for Expressions

We briefly discuss the consistency of the inference rules for expressions. The collection of inference rules in Table 4 is meant to be a tool that can aid in reasoning about a large variety of domains, each of which can be modelled using any of an equally broad variety of (potentially contradictory) logics. Thus, it would be impossible to define a reasonable notion of absolute consistency without limiting its applicability. Instead, we rely on a notion of relative consistency.

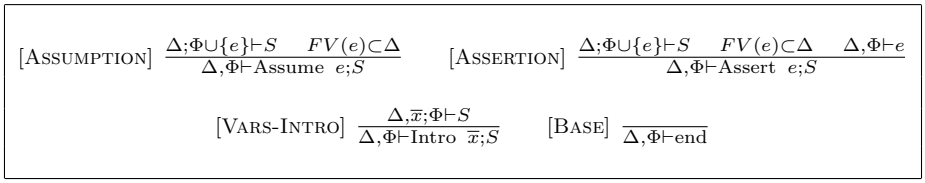
Definition 3.1. Let $\mathcal{L}$ be a consistent formal system. Suppose that there exists some $D_{\mathcal{L}}$ such that if e is deriviable under the inference rules when they are instantiated with $D = D_{\mathcal{L}}$, then e is deriviable in $\mathcal{L}$. We then say that our system is consistent relative to $\mathcal{L}$ under $D_{\mathcal{L}}$.

Thus, in a particular application, the basic technique that can be used to ensure the consistency of this collection of rules is the restriction of D, the range of substitutions. As a trivial example, we can consider the above rules under the restriction $D = \emptyset$. The inference rules for expressions then correspond to a strict subset of the inference rules for propositional logic.

Theorem 3.2. *Let $\mathcal{P}$ be the inference rules for propositional logic. Then the inference rules in Table 4 are consistent relative to $\mathcal{P}$ under $D = \emptyset$.*

Proof. This should be clear upon inspection: the rules for implication and conjunction are exactly the same as those in propositional logic, while [$\forall$-ELIM] and [$\exists$-INTRO] can never occur in a valid derivation because the premise $\overline{e} \subset D$ can never be met. Furthermore, [$\forall$-INTRO] cannot occur either, because statements

Table 5. Inference rules for a list of proof statements.



about the introduced quantified variables $\overline{x}$ must be derivable from an assumption context $\Phi \cap \{e | FV(e) \cap \overline{x} = \emptyset\}$, a context that cannot possibly contain any statements about any of the variables in $\overline{x}$ (such a derivation would only be possible through an application of the [$\forall$-Intro] rule, and we have already shown this is impossible). □

The relative consistency of the inference rules can be shown for other instantiations of D in a similar manner.

3.4. Inference Rules for Statements

A document consists of a sequence of statements $s_1; \ldots; s_n;$ end. Let $s; S$ denote that a statement s is followed by a list of statements S. The logical inference rules for statements are found in Table 5. Using an assumption statement, a user is able to introduce a new expression into the assumption context, so long as it has no free variables with respect to the variable environment Δ. An assertion statement is treated the same way as an assumption, with the added restriction that e must be derivable from the given assumption context Φ according to the inference rules for expressions in Table 4. In an introduction statement, the list of variables is simply added to the variable context.

4. Verification and Search Algorithms

Our formal representation's design puts a great deal of responsibility on the verification algorithm. At the very least, the algorithm must be able to determine which inference rule is being applied at each subsequent assertion. Such an algorithm is easy to define, but we believe that the algorithm must go further: the ultimate goal of an accessible verification system is that it must be capable of determining many of the implicit steps a human typically omits in a proof. Our approach in working towards this goal is to introduce a variety of new inference rules derivable from those in Table 4. This ensures the consistency of the algorithm, provides a uniform representation for search strategies, and allows for a relatively simple characterization for search depth bounds, as described in Section 4.5.

But first, we will distinguish the verification algorithm for statements from the verification algorithm for expressions that will be called as a subroutine. The verification algorithm for statements simply initializes empty environments $\Phi = \emptyset$

TABLE 6. Summary of basic verification algorithm.

$$
\begin{array}{rcl}
\text{verify}(\Delta, \Phi, e) & := & \text{is } e \in \Phi? \\
& & \text{or, does decompose}(\Delta, \Phi, e) \text{ succeed?} \\
& & \text{or, do there exist } \overline{e} \subset \Phi \text{ s.t. one of} \\
& & \quad [\text{IMPLIES-ELIM}],\ [\wedge\text{-ELIM-L}],\ [\wedge\text{-ELIM-R}], \\
& & \quad [\forall\text{-ELIM}], \text{ or } [\exists\text{-INTRO}] \text{ derive } e \text{ from } \overline{e}? \\
\\
\text{decompose}(\Delta, \Phi, e_1 \Rightarrow e_2) & := & \text{verify}(\Delta, \Phi \cup \{e_1\}, e_2) \\
\text{decompose}(\Delta, \Phi, e_1 \wedge e_2) & := & \text{verify}(\Delta, \Phi, e_1) \;\&\&\; \text{verify}(\Delta, \Phi \cup \{e_1\}, e_2) \\
\text{decompose}(\Delta, \Phi, \forall \overline{x}.e) & := & \text{verify}(\Delta \cup \overline{x}, \Phi \cap \{e \mid FV(e) \cap \overline{x} = \emptyset\}, e)
\end{array}
$$

and $\Delta = \emptyset$, and then processes a list of statements as determined by the rules in Table 5, extending environments as needed throughout the process. The only point at which the verification algorithm for expressions is called is when determining whether the premise $\Delta, \Phi \vdash e$ holds in the [ASSERTION] rule. Thus, we can now focus exclusively on the verification algorithm for expressions, whose input is (Δ, Φ, e), and whose output indicates whether it was able to construct a derivation concluding that $\Delta, \Phi \vdash e$.

4.1. Basic Verification Algorithm Implementation

Given the input, the basic verification algorithm attempts to apply each of the inference rules in Table 4. The only exception is the replacement of the [$\wedge$-INTRO] rule with the [$\wedge$-SEQ-INTRO] rule.

$$[\wedge\text{-SEQ-INTRO}]\frac{\Delta; \Phi \vdash e_1 \qquad \Delta; \Phi \cup \{e_1\} \vdash e_2}{\Delta; \Phi \vdash e_1 \wedge e_2}$$

This allows a user to author proofs at the expression level in the familiar manner under which each subsequent assertion is derived from previous assertions.

More specifically, the algorithm always tries to apply the [ASSUMPTION] rule, and then uses pattern matching to check if any of the rules [IMPLIES-INTRO], [$\wedge$-SEQ-INTRO], or [$\forall$-INTRO] might apply (we call this "decomposition"). In these cases, premises are checked by using a recursive call to the verification algorithm.[4] For the rules [IMPLIES-ELIM], [$\wedge$-ELIM-L], [$\wedge$-ELIM-R], [$\forall$-ELIM], and [$\exists$-INTRO], the algorithm checks the context Φ to see if the expression in question can be derived from one or more of the expressions found in Φ. The entire basic algorithm is summarized in Table 6. It should be apparent that this simple algorithm can successfully verify any derivable formula, as long as each and every step of the derivation is presented to the algorithm in sequence.

[4]We discuss efficiency and termination in Section 4.5

TABLE 7. "Shortcut" logical inference rules for searching.

$$[\text{SEARCH-BASE}]\ \frac{\theta(e_0) \equiv e}{\Delta;\Phi \vdash \theta(e_0) \Rightarrow e} \quad [\text{SEARCH-}\wedge\text{-L}]\ \frac{\Delta;\Phi \vdash \theta(e_1) \Rightarrow e}{\Delta;\Phi \vdash \theta(e_1 \wedge e_2) \Rightarrow e} \quad [\text{SEARCH-}\wedge\text{-R}]\ \frac{\Delta;\Phi \vdash \theta(e_2) \Rightarrow e}{\Delta;\Phi \vdash \theta(e_1 \wedge e_2) \Rightarrow e}$$

$$[\text{SEARCH-IMPLIES}]\ \frac{\Delta;\Phi \vdash \theta(e_2) \Rightarrow e \quad \Delta;\Phi \vdash \theta(e_1)}{\Delta;\Phi \vdash \theta(e_1 \Rightarrow e_2) \Rightarrow e} \qquad [\text{SEARCH-}\forall]\ \frac{\Delta;\Phi \vdash \theta(e_0) \Rightarrow e \quad \theta' = \theta - \overline{x}}{\Delta;\Phi \vdash \theta'(\forall \overline{x}.e_0) \Rightarrow e}$$

4.2. Generalized Substitution Search Algorithm

Denote by $\equiv$ syntactic and alphabetic equivalence of expressions. Note that

$$\begin{aligned} \theta(e_1 \wedge e_2) &\equiv \theta(e_1) \wedge \theta(e_2) \\ \theta(e_1 \Rightarrow e_2) &\equiv \theta(e_1) \Rightarrow \theta(e_2), \end{aligned}$$

and that so long as $dom(\theta) \cap \overline{x} = \emptyset$,

$$\theta(\forall \overline{x}.e) = \forall \overline{x}.\theta(e).$$

Denote by $\theta' = \theta - \overline{x}$ the fact that for all $y \in dom(\theta') \cap dom(\theta)$, $\theta(y) \equiv \theta'(y)$, that $dom(\theta') = dom(\theta) - \overline{x}$, and that $\overline{x} \subset dom(\theta)$. These equations, along with the inference rules for expressions in Table 4, can now be used to derive the "shortcut" inference rules in Table 7.

The basic verification algorithm can now be extended. Given (Δ, Φ, e), for every $e' \in \Phi$, one can ask if it is possible to construct a derivation concluding that $\Delta, \Phi \vdash \theta_0(e') \Rightarrow e$, where $dom(\theta_0) = \emptyset$.

$$[\text{SEARCH}]\frac{e' \in \Phi \quad dom(\theta_0) = \emptyset \quad \Delta; \Phi \vdash \theta_0(e') \Rightarrow e}{\Delta; \Phi \vdash e}$$

This new [SEARCH] rule can be checked at any point in the algorithm at which the [ASSUMPTION] rule is checked.

4.2.1. Limitations. In some cases, this search algorithm is not sufficient, as it is not possible to construct a substitution that resolves all free variables in the left-hand side of an implication, as the example below illustrates. We assume that $D = \mathbb{N}$ for the purposes of this example.

Assume	for any $x, y, z,\ x < y$ and $y < z$ implies $x < z$.
Assume	$0 < 1$.
Assume	$1 < 2$.
Assert	$0 < 2$.

Notice that even if a substitution is obtained that can handle all free variables in the expression $x < z$ (mapping x to 0 and z to 2), y is not in the substitution's domain, so it is not possible to simply verify $x < y$ and $y < z$ under the substitution. We avoid this limitation in *some* cases by preprocessing all expressions before verification, pushing universally quantified variables that do not appear on

TABLE 8. "Shortcut" logical inference rules for existentially quantified expressions.

$$[\exists\text{-SEQ-AND-INTRO}]\ \frac{\Delta;\Phi\vdash\exists\overline{x}.e_1 \qquad \Delta,\overline{x};\Phi\cup\{e_1\}\vdash e_2}{\Delta;\Phi\vdash\exists\overline{x}.e_1\wedge e_2} \qquad [\exists\text{-SEQ-AND-ELIM-R}]\ \frac{\Delta;\Phi\vdash\exists\overline{x}.e_1\wedge e_2}{\Delta;\Phi\vdash\exists\overline{x}.e_1}$$

the right-hand sides of implications down into the left-hand sides. In the example, the first assumption would then be translated to the following:

Assume for any x, z, (there exists y such that $x < y$ and $y < z$) implies $x < z$.

We found that this adjustment is sufficient for handling many real situations, as human authors seldom construct statements that preclude this transformation.

4.3. Verification of Existential Expressions

Our algorithm attempts to verify existential expressions in an additional, special manner that allows users to reason "under" an existential quantifier, according to the rules in Table 8. This is typically the only option for a user, as the inference rules contain no elimination rule for existential quantifiers. We have found that these two rules are sufficient for constructing succinct proofs for a variety of examples involving existential quantifiers, one of which is the example presented in Table 1.

4.4. Other Inference Rules

Our implementation includes a variety of additional inference rules corresponding to common axioms and normal forms on expressions that contain the constant symbols available in the language. Many of these could be included as a separate library of results written in the language, but including them within the implementation improves performance substantially. A few, such as normalization algorithms for arithmetic equations, would be more difficult to represent externally without incurring serious performance costs.

4.5. Resource Limitations and Complexity

When the verification algorithm attempts to apply a particular inference rule, the rule's premises must be verified by a recursive call. Thus, unless the algorithm is bounded in some manner, it may spend an exponential amount of time trying to verify a proof, or it may not terminate at all. In order to address this issue, the verification algorithm maintains an integer depth parameter, and every inference rule is coupled with additional information that indicates whether the depth parameter should be decremented when the algorithm attempts to verify the premises of the rule. Introduction rules do not require that the depth parameter be decremented, because the expressions in the premises of these rules are strictly smaller. In all other cases, the parameter is decremented. If the parameter ever reaches 0, verification fails.

Thus, given an assumption context Φ in which e^* is the *largest* expression, one can approximate the verification algorithm's complexity on some input (Δ, Φ, e) by the expression

$$O\Big((|e| \cdot |e^*| \cdot |\Phi|)^d\Big),$$

where $|e|$ denotes the size of an expression, and d denotes the depth parameter. We have found that on a modern 3.0GHz processor with 1GB of RAM, $d = 6$ leads to acceptable performance (at most a few seconds for a one- or two-page proof) and relatively succinct proofs for many of the examples we have constructed.

5. Related Work

In recent work [3], the ΩMEGA proof verifier [11] has been integrated with the scientific text editor TEX_{MACS}. As in our work, the authors defined a concrete representation that consists of selected English phrases and LaTeX syntax. The authors also advocate an interactive verification user experience in which the user makes modifications to a proof document, and the verifier performs a search to repair incomplete proofs. The general search technique framework employed by the authors is similar to ours. However, one important difference is in our view of search heuristics: we view them as an essential feature that ensures that the concrete representation is forward compatible and contains no helpful annotations for any particular verifier. Furthermore, as we stated earlier, we believe that integration of a system with a *particular* text editor will often serve as an obstacle to the adoption of that verification system. Finally, an important difference is that we emphasize a more lightweight approach that allows the user to choose to verify only parts of her document, to use any underlying logic she wishes to use, and to introduce and immediately use high-level statements of results in particular domains using a familiar and easy to understand syntax.

The MathLang project [6] is an extensive, long-term effort that aims to make natural language an input method for mathematical arguments and proofs. Natural language arguments are converted into a formal grammar (without committing to any particular semantics), and the participants are currently working on ways to convert this representation into a fully formalized logic. The MathLang project is focused primarily on mathematical texts, and representative texts from different areas are used as a guide in determining the direction of future research in the project. However, the project does not address several of the issues we raise. Once again, a user wishing to take advantage of this system must use a specialized editor associated with the MathLang project. Furthermore, while the complex parsing algorithm may be able to handle a larger variety of natural language statements, this actually makes learning the capabilities and limitations of the parsing algorithm a more difficult task. This work also does not recognize and exploit the need for search heuristics in simplifying the concrete formal representation.

More generally, there 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 [14]. Some of these tools provide a way to construct proofs by induction, such as Coq [8], PVS [7], and Isabelle [9, 10]. More specifically, formal representation and verification systems include Isabelle/Isar [13] and Mizar [12]. 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. However, in both of these systems, no attempt is made to provide intuitive proof search features in response to a forward-compatible representation design that necessarily excludes any information that explicitly aids the underlying verifier. Furthermore, these systems are not designed to be lightweight, but to guarantee consistency of arguments with respect to particular logic(s).

Yet other systems allow the construction of static models using first order logic while only providing partial confidence in correctness, such as Alloy [5]. This modelling language allows a user to formally specify a set of constraints, and then perform a search for counterexamples in a finite space. Alloy is intentionally designed to allow partial and incomplete specifications. This lightweight approach is demonstrably useful and inspired the lightweight emphasis of our own design. In a way, our system also performs a limited state space search, but differs in that it guarantees relative consistency by only returning false negatives, and no false positives. Also, the concrete syntax of Alloy is inspired by popular programming languages, while our syntax is based on mathematical notation and natural language.

6. Conclusions

We have shown that it is possible to construct a formal representation and verification system in a manner that addresses and mitigates some of the common disincentives to using such systems. A verification system can provide a familiar, friendly syntax that is independent of the strategies used by the underlying verifier, and a simple verifier can be augmented in a uniform and consistent manner with a variety of heuristic search strategies that further enhance usability.

There are many obvious avenues for further work. While we do not believe that the basic structure of the representation should be augmented much further, it should be possible to load new syntactic constants from a configuration file (this is easily done with the Parsec library), or to allow the user to define new constants using a new kind of statement. It should be possible to extend the logic itself with disjunction and a limited form of negation without compromising the simplicity of the proof and expression language. For the verification algorithm, a variety of additional verification strategies are planned for specific domains, and we hope that it will eventually be possible to encode these strategies within the language itself idiomatically, with minimal extensions to the syntax. For example, it should be possible to define pure, finitely computable functions by using the existing syntax, and such definitions should be detected and recognized by the verifier as

functions. It is also our intention to more extensively test our hypothesis that our formal representation and verification system is more accessible by deploying it within the classroom, and by assembling libraries of results in specific domains.

References

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 3–22, London, UK, 2000. Springer-Verlag.

[2] H. J. M. M. D. J. P. Leijen. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

[3] D. Dietrich, E. Schulz, and M. Wagner. Authoring verified documents by interactive proof construction and verification in text-editors. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 398–414, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] K. Grue. The layers of logiweb. In *Calculemus '07 / MKM '07: Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants*, pages 250–264, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[6] F. Kamareddine and J. B. Wells. Computerizing mathematical text with mathlang. *Electron. Notes Theor. Comput. Sci.*, 205:5–30, 2008.

[7] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. 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.

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

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

[10] L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and Its Applications*. MIT Press, 1997.

[11] J. H. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof development with omega: sqrt(2) is irrational. In *LPAR*, pages 367–387, 2002.

[12] A. Trybulec and H. Blair. Computer assisted reasoning with mizar. In *Proc. of the 9th IJCAI*, pages 26–28, Los Angeles, CA, 1985.

[13] M. Wenzel and L. C. Paulson. Isabelle/isar. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 41–49. Springer, 2006.

[14] F. 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.

Andrei Lapets
Department of Computer Science
Boston University
111 Cummington St.
Boston, MA 02215
USA
e-mail: lapets@bu.edu