

Designing accessible 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 by drawing on our own reasoning as well as that of existing work: (1) a simple, familiar, and conventional concrete syntax that is independent of any environment, application, or verification strategy; (2) extensive native support for at least basic constructions and practices related to logical reasoning, arithmetic, naive set theory, algebraic manipulation, and evaluation of simple functions; and (3) a lightweight underlying structure that reduces workload and entry costs by allowing users to employ features selectively and by automatically managing multiple logical systems without requiring a user’s involvement or technical knowledge.

We demonstrate the feasibility of satisfying these criteria by presenting our own formal representation and verification system. Our system’s simple concrete syntax overlaps with English, L^AT_EX and MediaWiki markup. The system’s underlying verifier demonstrates our main contributions. It relies on heuristic search techniques, data structures, syntactic analyses, and normal forms to make the proof authoring process more manageable and consistent with prevailing practices. Furthermore, its underlying logic consists of a symbolic template that enables lightweight verification but can also be used to automatically manage multiple logical systems through straightforward syntactic restrictions.

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 summarize and discuss our own understanding of some potential obstacles and disincentives to using existing, well-established formal representation and verification systems. We believe that many of these obstacles 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. Our discussion echoes and further develops observations that motivated other recent projects with similar goals, such as the *Tutch* proof checker [2], the *Scunak* mathematical assistant system [3], and the *ForTheL* language and *SAD* proof assistant [18]. In our discussion, 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 or overly complex syntax and/or environment. In many cases, the user must familiarize herself with a potentially rich 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 [7]. 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. Our observations are shared by

the designers of Scunak [3], who refer to the need for “naturalness” in a system’s concrete representation.

One popular approach to the design of interfaces for proof authoring and verification systems is embodied in interactive theorem provers, such as those designed for Coq [12] and Isabelle/Isar [19]. However, as has been observed before [2], this is actually an inconveniently rigid framework. The user is required to learn how to direct the interactive system, cannot easily “jump around” while constructing a proof, and cannot resort to a lightweight approach under which only some parts of the proof are formal and correct.

Unnatural, cumbersome structure. Many systems, such as Coq [12], Isabelle/Isar [19], Mizar [17], and PVS [11], allow arguments to be built up from the fundamental principles and entities of particular logics. Wide-ranging libraries of other common concepts must then be assembled from these basic building blocks. In its current form, this practice has some drawbacks.

While it may lead to simpler and more elegant theoretical systems, there is no reason that a system intended for practical use should not provide *native* support for many ubiquitous concepts and practices, including at least arithmetic, naive set theory, common algebraic manipulations, and simple functions (and if we permit ourselves to be ambitious, perhaps even graph theory and algorithms). The practice of relying on the underlying logic of a system for these features often results in what are intuitively fundamental concepts, such as algebraic manipulations or simple functions, being represented using some other fundamental but entirely unrelated construct, such as relations and sets. Recent work in this area reveals that some sympathize with this reasoning. For example, the Scunak system [3] includes native support for some concepts in set theory, and the designers of `Tutch` explicitly recognize in their conclusions that the ability to do algebra by chaining lists of equations together is an essential feature for future proof systems.

Another issue is that 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 syntax error detection, unbound variable detection, or type checking, and does not wish to invest any extra effort to enjoy additional benefits. Sometimes a user does not have confidence in arguments based on the fundamental principles of a logic because she is not certain the logic is an adequate model for her purposes. This is a common concern in research on the safety of cryptographic protocols [1]. Recent work reveals that others are beginning to acknowledge this concern, offering a notion of correctness that deals only with syntax and bound variables called “ontological correctness”

[18], or demonstrating that a type system can provide some limited verification capabilities [3]. This progress presents the opportunity to go further by providing even more flexibility to the user in the form of “lightweight” verification methods.

Many established verification tools, such as in Coq, Isabelle/Isar, Mizar, and PVS, do not allow such selections to be made easily. It is certainly possible within these systems to introduce user-defined axioms, but the interfaces do not treat this as a primary and essential activity. On the other end, systems like the Alloy modelling language [6] do encourage this sort of approach, but once the commitment has been made to using only the benefits provided by the chosen system, there is no possibility of introducing a more rigorous approach in the future using a *compatible* representation.

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 such an approach, facets of which are already being investigated in existing work [3, 2], is further demonstrated by the formal representation and verification system we present in this work. We enumerate the main design principles and discuss how they manifest themselves in our system.

Familiar, simple concrete syntax. We adopt a familiar and simple 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. In this regard we follow the designers of other systems, such as Scunak [3], that have also recognized the value of allowing systems to deal with \LaTeX directly. Our 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 with \LaTeX and English only needs to learn three or four simple syntactic constructs and conventions to begin writing formal arguments immediately. This characteristic of our system reflects the austere design of the Tutch system [2], which has only four basic constructs. Because our representation itself is no more complex than \LaTeX (and actually much simpler), it is not necessary to use any particular editor or environment, and the user has the flexibility of constructing the proof in any order. 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 allowing users to verify their formal arguments as they are writing or editing them simply by clicking a button on the edit page.

¹<http://www.safre.org>

Implicit arguments/explicit results, and native support. 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*. This is a principle to which the designers of the Scunak system [3] refer 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” [2].

Another benefit of such a representation is that it is *forward compatible*: special annotations that might 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. This very approach can be seen in the design of several features of our system, described in Section 4.

As we have already mentioned, it is essential that there be native support for other common practices that include arithmetic, algebraic manipulations, set theory, and application of simple functions. Thus, our system provides a basic assortment of built-in constants and appropriate common evaluation, normalization, and inference rules. We believe that committing to specific rules for constants improves the accessibility of the system to an extent that overcomes any loss of flexibility. We have found that these features work together to further support the “implicit arguments, explicit results” approach to proof authoring.

Lightweight design and automatic management of logical systems. Lightweight design need not be limited to a particular component, but should be visible throughout the various components of a system. We try to move further towards this goal by offering a flexible concrete syntax that allows the user to mix informal writing and verifiable content, as well as the option of using an underlying “lightweight” logic that has very few restrictions. However, lightweight design is not exclusively a passive principle that consists of loosening restrictions. A system can be lightweight from the user’s perspective while still actively anticipating what the user is trying to do by using heuristic techniques and automatically tightening the user’s work. Systems such as Ω MEGA [16] arguably demonstrate this idea at the level of individual proof steps by attempting to do proof reconstruction on a formal argument after a user has made changes to it. We attempt to lift this idea to a more abstract level: our system tries to recognize when a user is restricting herself only to a certain logical system (such as propositional logic or first order logic) and re-assures the user preemptively of this fact if she is successful in doing so. We

believe this is valuable because most users (including professional mathematicians) do not have the interest (or the expertise) to manage or choose appropriate logical systems when doing their work. And more generally, the interesting question for a user is not always which individual law or assumption applies, but which logical *family* of rules is necessary to prove a result.

A lightweight design facilitate a formal reasoning process that is both more manageable and gradual. Even in the absence of a supporting library for her domain, a user can immediately begin authoring formal arguments and enjoying the benefits of detection of simple errors. At a later point in time, the user may introduce additional detail that increases confidence in her arguments, slowly constructing a library of results. 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 she 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, the logical inference rule template, and how the template can be instantiated to represent particular logical systems. Section 4 describes the basic verification algorithm, shows how it is extended with heuristic proof search capabilities, and gives an overview of the variety of additional verification capabilities that support the user's authoring process. We review recent related work and provide some wider context 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 L^AT_EX. The actual concrete syntax can be seen in Table 2. It is worth noting that the entire L^AT_EX 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

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, $m \cdot \sqrt{2} = n$,

$$(m \cdot \sqrt{2})^2 = n^2,$$

$$m^2 \cdot \sqrt{2}^2 = n^2,$$

$$m^2 \cdot 2 = n^2,$$

$$n^2 = m^2 \cdot 2,$$

$$n^2 = 2 \cdot m^2, \text{ and so, } n^2 \text{ is even. Thus, } n \text{ is even.}$$

Furthermore, there exists $j \in \mathbb{Z}$ s.t. $n = 2 \cdot j$,

$$n^2 = (2 \cdot j)^2,$$

$$n^2 = 2^2 \cdot j^2,$$

$$n^2 = 4 \cdot j^2,$$

$$2 \cdot m^2 = 4 \cdot j^2,$$

$$m^2 = 2 \cdot j^2,$$

$$m^2 \text{ is even and } m \text{ is even.}$$

Thus, we have a contradiction.

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, case distinctions, and so forth. This reduces the amount of new, specialized syntax and semantics a user must learn when employing

TABLE 2. Example of a verifiably consistent argument.

We introduce some assumptions about integers and rational numbers.

\vbeg

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

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

Assume for any $i \in \mathbb{Z}$, $\{i \text{ is even}\}$ 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. $\{n \text{ and } m \text{ have no common factors}\}$ 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. $\{i \text{ is even}\}$, $\{j \text{ is even}\}$, and $\{i \text{ and } j \text{ have no common factors}\}$ then $\{\text{we have a contradiction}\}$.

\vend

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

\vbeg

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

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, $\{n^2 \text{ is even}\}$. Thus, $\{n \text{ is even}\}$.

Furthermore, there exists $j \in \mathbb{Z}$ s.t. $n = 2 \cdot j$,

$$n^2 = (2 \cdot j)^2,$$

$$n^2 = 2^2 \cdot j^2,$$

$$n^2 = 4 \cdot j^2,$$

$$2 \cdot m^2 = 4 \cdot j^2,$$

$$m^2 = 2 \cdot j^2,$$

$$\{m^2 \text{ is even}\} \text{ and } \{m \text{ is even}\}.$$

Thus, $\{\text{we have a contradiction}\}$.

\vend

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 \vee , \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

2.2. Parsing

The parser for the concrete syntax was constructed in Haskell using the Parsec parser combinator library [9], 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.

3. Abstract Syntax and Logical Inference Rules

We present the abstract syntax for statements, expressions, and constants. We also define the inference rules for expressions, which are used to determine what kinds of expressions represent “true” assertions with respect to some logical systems. Finally, we define the inference rules for statements, which are used to verify a mathematical document comprised of a list of statements.

²www.mediawiki.org

³www.safre.org

TABLE 3. Abstract syntax.

expressions	e	$::=$	$c \mid x \mid e_1 e_2 \mid (e_1, \dots, e_n)$
			$\mid e_1 \Rightarrow e_2 \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e$
			$\mid \forall \bar{x}.e \mid \exists \bar{x}.e$
statements	s	$::=$	$\text{Assume } e \mid \text{Assert } e \mid \text{Intro } \bar{x}$

3.1. Abstract Syntax

In Table 3 we summarize the abstract syntax for our representation and verification system. Let X be the set of all variables. We denote by x a single variable, and by \bar{x} a vector of variables. We sometimes also denote by \bar{x} the *set* of variables found in \bar{x} (thus, $\bar{x} \subset X$). 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 and is used to represent a variety of constructions. Table 4 presents a more detailed look at a portion of the abstract syntax corresponding to common constants used in arithmetic and set theory. Whenever there is a possibility of confusion or ambiguity, we use a tilde to distinguish a symbolic constant within our abstract syntax from the actual mathematical object (e.g. $\tilde{+}$ or $\tilde{\mathbb{Z}}$ for a constant in our abstract syntax, and $+$ or \mathbb{Z} for the mathematical object).

We explain in more detail how some common mathematical constructions are represented within the abstract syntax.

English phrase predicates. English phrases acting as predicates can have 0 or more arguments. An English phrase predicate is represented 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 $\backslash p\{\$p\$ \text{ is a path in } \$G\}$ is represented using the list

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

and the entire expression is represented as

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

Mathematical operators. Common mathematical operators (such as those in arithmetic and basic set theory) are treated as constants. If an operator has an arity of 2 or above, it is by convention applied to a tuple of expressions. Thus, the input $\$2 + 2\$$ is represented as

$$\tilde{+} (2, 2).$$

TABLE 4. Abstract syntax for constants.

phrases	p	::=	English phrase predicates
numeric literals	q	\in	\mathbb{Q}
constants	c	::=	p q $\tilde{+}$ $\tilde{-}$ $\tilde{\cdot}$ $\tilde{/}$ $\tilde{\text{mod}}$ $\tilde{\text{pow}}$ $\tilde{\text{log}}$ $\tilde{\text{max}}$ $\tilde{\text{min}}$ $\tilde{\sqrt{\quad}}$ $\tilde{<}$ $\tilde{\leq}$ $\tilde{>}$ $\tilde{\geq}$ $\tilde{\cup}$ $\tilde{\cap}$ $\tilde{\times}$ $\tilde{\rightarrow}$ $\tilde{[\quad]}$ $\tilde{\lceil \quad \rceil}$ $\tilde{\{ \quad \}}$ $\tilde{\in}$ $\tilde{\subset}$ $\tilde{=}$ $\tilde{\mathbb{N}}$ $\tilde{\mathbb{Z}}$ $\tilde{\mathbb{Q}}$ $\tilde{\mathbb{R}}$ (c_1, \dots, c_n) $\{c_1, \dots, c_n\}$

Explicit sets. The operator $\tilde{\{ \}}$ is applied to a tuple expression to represent an explicit set within the expression syntax. For example, the set $\{1, 2, 3\}$ is represented as

$$\tilde{\{ \}} (1, 2, 3).$$

Explicit ranges for quantified variables. Any instance of concrete syntax of the form

$$\forall x_1 \tilde{\in} e_1, \dots, x_n \tilde{\in} e_n. e$$

is converted into abstract syntax of the form

$$\forall x_1, \dots, x_n. ((x_1 \tilde{\in} e_1) \wedge \dots \wedge (x_n \tilde{\in} e_n)) \Rightarrow e.$$

This also applies to the existential quantifier. This process is performed variable by variable starting from the innermost variable because the user might use a variable x_i in any expression e_j where $j > i$. In such cases, the expression is transformed into nested quantifiers, i.e.

$$\forall x_1, \dots, x_{j-1}. ((x_1 \tilde{\in} e_1) \wedge \dots \wedge (x_{j-1} \tilde{\in} e_{j-1})) \Rightarrow (\forall x_j, \dots, x_n. ((x_j \tilde{\in} e_j) \wedge \dots \wedge (x_n \tilde{\in} e_n)) \Rightarrow e)$$

Chained (in)equalities of binary relations. A common practice when dealing with the mathematical relations represented by constants such as $\tilde{\leq}$, $\tilde{\geq}$, $\tilde{<}$, $\tilde{>}$, $\tilde{=}$, $\tilde{\subset}$ is to chain equations or inequalities together. For example, one might write

$$1 \tilde{\leq} 2 \tilde{\leq} 3.$$

Our parser supports this concrete syntax, and converts any explicit chain appropriately. In other words, if for all i , $\otimes_i \in \{\tilde{\leq}, \tilde{\geq}, \tilde{<}, \tilde{>}, \tilde{=}, \tilde{\neq}\}$, then any expression of the form

$$e_1 \otimes_1 e_2 \otimes_2 \dots \otimes_{n-2} e_{n-1} \otimes_{n-1} e_n$$

is converted into an expression of the form

$$(e_1 \otimes_1 e_2) \wedge (e_2 \otimes_2 e_3) \wedge \dots \wedge (e_{n-1} \otimes_{n-1} e_n).$$

A user can use grouping symbols (parentheses or braces) if they wish to write an expression like $(1\tilde{<}2)\tilde{=}(1\tilde{<}2)$, in which a statement about the *result* of $(1\tilde{<}2)$ is made. Currently, it is only possible to extend this capability to new operators by adding them to an appropriate list within the system’s configuration files.

3.2. Inference Rule Template for Expressions

Denote by \equiv syntactic and alphabetic equivalence of expressions. We define $FV(e)$ to be the collection of free variables in an expression e . We also define substitutions on expressions. For any vector of variables \bar{x} and any vector of expressions \bar{e} , we denote by $[\bar{x} \mapsto \bar{e}]$ the operator that performs a capture-avoiding substitution in the usual manner on any expression (replacing each x in \bar{x} with the corresponding e' in \bar{e}). If $\theta = [\bar{x} \mapsto \bar{e}]$, then $\theta(e)$ and $e[\bar{x} \mapsto \bar{e}]$ both denote the application of this operator on an expression e . For a substitution θ we write $\text{dom}(\theta)$ to denote the set of variables bound within the substitution and $\text{ran}(\theta)$ to denote the set of expressions to which the variables in θ are mapped. Given two expressions e, e' and some variables \bar{x} , we define $\text{match}(\bar{x}, e, e')$ to be the substitution that syntactically unifies (makes syntactically equivalent) e and e' by appropriately substituting any free occurrences in e of the variables from \bar{x} . Note that this is one-sided: the substitution $\theta = \text{match}(\bar{x}, e, e')$ is such that $\theta(e) \equiv e'$ but it may be that $e \not\equiv \theta(e')$. Obviously, $\text{match}(\bar{x}, e, e')$ can be undefined.

The variable Δ will always denote a context of bound variables that maps variables to a fixed set of sorts, 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 bound in Δ , that all expressions in Φ also have no free variables other than those bound in Δ , and that we consider e to represent a “true” statement in some sense.

We define in Table 5 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 syntactically-defined subset of admissible expressions, and can thus be used to represent different logics. However, they can only be used to represent logics in which the sort of a variable within an expression can always be determined using syntax alone. While this may seem like a serious limitation, it is important to note that we view this capability merely as a bookkeeping convenience that allows us to keep track of which logical system a user is trying to employ, and *not* as a restriction on the system. When verifying an expression, a verification algorithm attempts to infer it under *many* possible instantiations of the template. As a last resort, it tries the trivial instantiation under which all variables are of

TABLE 5. Template for logical inference rules for expressions.

$[\text{CONTEXT}] \frac{e \in \Phi \quad FV(e) \subset \Delta}{\Delta; \Phi \vdash e}$		
$[\text{IMPLIES-INTRO}] \frac{\Delta; \Phi \cup \{e_1\} \vdash e_2}{\Delta; \Phi \vdash e_1 \Rightarrow e_2}$	$[\text{IMPLIES-ELIM}] \frac{\Delta; \Phi \vdash e_1 \Rightarrow e_2 \quad \Delta; \Phi \vdash e_1}{\Delta; \Phi \vdash e_2}$	
$[\wedge\text{-INTRO}] \frac{\Delta; \Phi \vdash e_1 \quad \Delta; \Phi \vdash e_2}{\Delta; \Phi \vdash e_1 \wedge e_2}$	$[\wedge\text{-ELIM-L}] \frac{\Delta; \Phi \vdash e_1 \wedge e_2}{\Delta; \Phi \vdash e_1}$	$[\wedge\text{-ELIM-R}] \frac{\Delta; \Phi \vdash e_1 \wedge e_2}{\Delta; \Phi \vdash e_2}$
$[\vee\text{-INTRO-L}] \frac{\Delta; \Phi \vdash e_1}{\Delta; \Phi \vdash e_1 \vee e_2}$	$[\vee\text{-INTRO-R}] \frac{\Delta; \Phi \vdash e_2}{\Delta; \Phi \vdash e_1 \vee e_2}$	$[\vee\text{-CASE}] \frac{\Delta; \Phi \vdash e_1 \Rightarrow e_0 \quad \Delta; \Phi \vdash e_2 \Rightarrow e_0}{\Delta; \Phi \vdash (e_1 \vee e_2) \Rightarrow e_0}$
$[\text{NEGATION}] \frac{\Delta; \Phi \vdash \neg e}{\Delta; \Phi \vdash e}$	$[\text{CONTRADICTION}] \frac{\Delta; \Phi \vdash e_1 \Rightarrow e_2 \quad \Delta; \Phi \vdash e_1 \Rightarrow \neg e_2}{\Delta; \Phi \vdash \neg e_1}$	
$[\forall\text{-INTRO}] \frac{\Delta, \bar{x} \mapsto \sigma(e, \bar{x}); \Phi \cap \{e \mid FV(e) \cap \bar{x} = \emptyset\} \vdash e}{\Delta; \Phi \vdash \forall \bar{x}. e}$		
$[\forall\text{-ELIM}] \frac{\Delta; \Phi \vdash \forall \bar{x}. e \quad \bar{e} = \bar{x} \quad \xi(\Delta, \bar{e}) = \sigma(\bar{x}, e) \quad FV(\bar{e}) \subset \Delta}{\Delta; \Phi \vdash e[\bar{x} \mapsto \bar{e}]}$		
$[\exists\text{-INTRO}] \frac{\Delta; \Phi \vdash e[\bar{x} \mapsto \bar{e}] \quad \bar{e} = \bar{x} \quad \xi(\Delta, \bar{e}) = \sigma(\bar{x}, e) \quad FV(\bar{e}) \subset \Delta}{\Delta; \Phi \vdash \exists \bar{x}. e}$		

the same sort and all expressions are admissible. This last resort corresponds to a kind of “lightweight” verification strategy.

A straightforward way to represent a particular instantiation of the inference rules is to define four sets $s^{\mathcal{L}}$, $t^{\mathcal{L}}$, $a^{\mathcal{L}}$, $e^{\mathcal{L}}$ of the space of syntactically well-formed expressions (corresponding to *sorts*, *terms*, *atoms*, and *sentences* as defined in popular presentations of logic [8]). Examples can be found in Tables 6, 7, and 8 in Section 3.3, where we discuss how these instantiations ensure the consistency of the inference rules with respect to particular logical systems. Notice that occurrences of variables within the syntax definitions (represented by x_{sort}) are labelled by the syntactically-determined sort of all variables found in that position.

The inference rules interact with the parameters $s^{\mathcal{L}}$, $t^{\mathcal{L}}$, $a^{\mathcal{L}}$, $e^{\mathcal{L}}$ in two ways. First, an additional premise is implied for *every* inference rule in Table 5:

$$\frac{\dots \quad e \in e^{\mathcal{L}}}{\Delta; \Phi \vdash e}$$

We omitted this rule for the sake of clarity. It ensures that only sentences as defined by the parameter $e^{\mathcal{L}}$ can be inferred. Second, two functions σ and ξ are used to determine the sorts of variables and expressions. The function σ takes an expression and a variable, and returns the syntactically-determined sort of that variable within the expression. If it cannot determine a sort or if there is an inconsistency in the way the variable is used, the function is undefined. The function ξ takes a context Δ and an expression e , and determines its sort within

that context by checking which syntactic definition $t^{\mathcal{L}}$ the expression respects and using Δ to determine the sort of variables within the expression. If it cannot determine a sort or there is a contradiction, the function is undefined. The symbols σ and ξ are overloaded to also represent the vector versions of these functions (corresponding to element-wise mappings of $\sigma(e)$ on a vector of variables and $\xi(\Delta)$ on a vector of expressions).

Some of the inference rules are worth addressing. The $[\forall\text{-INTRO}]$ rule discards from the context any formulas that contain (as free) the variables being introduced by the universal quantifier. The reader may wonder why α -conversion could not address this problem. While this may be a reasonable solution when the expressions in question are only viewed by a machine in their final form, it makes less sense when we consider the perspective of a user employing step-by-step forward reasoning. If the user wishes to employ an assumption involving a variable that is currently within the context, she is unlikely to re-introduce this variable by quantifying over it universally; in fact, if she does do so, it is likely to be an error. Nevertheless, it is worth noting that our definition of this rule is based on our preferences, and alternative approaches are not ruled out for the future.

Note that the first premise of $[\forall\text{-INTRO}]$ not only requires that the syntactically-determined sorts of the newly-introduced variables be recorded within Δ , it requires that the variable have the same sort in *every* subexpression of e in which it appears (otherwise, σ is undefined). Both the $[\forall\text{-ELIM}]$ and $[\exists\text{-INTRO}]$ rules require that the sorts of the expressions \bar{e} match the sorts of the variables \bar{x} that they will replace. This ensures that any attempt to substitute variables for expressions respects the sorts defined within a logical system.

3.3. Consistency of the Inference Rules for Expressions

It is desirable to establish the consistency of the inference rules for expressions to aid users looking for a strict verification of their arguments with respect to a particular logical system. However, the collection of inference rules in Table 5 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. Consequently, it would be inadvisable and even impossible to define an absolute notion of consistency without limiting the applicability of the entire system and any verifier supporting it. 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 quadruple $s^{\mathcal{L}}, t^{\mathcal{L}}, a^{\mathcal{L}}, e^{\mathcal{L}}$ such that if an expression e is derivable under the inference rules in Table 5 when they are instantiated with this quadruple, then e is derivable in \mathcal{L} . We then say that our system is consistent relative to \mathcal{L} under $s^{\mathcal{L}}, t^{\mathcal{L}}, a^{\mathcal{L}}, e^{\mathcal{L}}$.

Thus, in a particular application, the basic technique that can be used to ensure the consistency of this collection of rules is an appropriate definition of $s^{\mathcal{L}}, t^{\mathcal{L}}, a^{\mathcal{L}}, e^{\mathcal{L}}$. It is worth briefly noting that the last resort, “lightweight” instantiation

TABLE 6. Abstract syntax subset for propositional logic (PL).

sorts	s^{PL}	=	\emptyset (no sorts exist)
terms	t^{PL}	=	\emptyset (no valid terms exist)
atoms	a^{PL}	::=	p
sentences	e^{PL}	::=	$a^{\text{PL}} \mid e_1^{\text{PL}} \Rightarrow e_2^{\text{PL}} \mid e_1^{\text{PL}} \wedge e_2^{\text{PL}} \mid e_1^{\text{PL}} \vee e_2^{\text{PL}} \mid \neg e^{\text{PL}}$

TABLE 7. Abstract syntax subset for first order logic (FOL)

sorts	s^{FOL}	=	elem
terms	$t_{\text{elem}}^{\text{FOL}}$::=	x_{elem}
atoms	a^{FOL}	::=	$p \mid p(t_{\text{elem}}^{\text{FOL}}, \dots, t_{n_{\text{elem}}}^{\text{FOL}})$
sentences	e^{FOL}	::=	$a^{\text{FOL}} \mid e_1^{\text{FOL}} \Rightarrow e_2^{\text{FOL}} \mid e_1^{\text{FOL}} \wedge e_2^{\text{FOL}} \mid e_1^{\text{FOL}} \vee e_2^{\text{FOL}}$ $\mid \neg e^{\text{FOL}} \mid \forall \bar{x}. e^{\text{FOL}} \mid \exists \bar{x}. e^{\text{FOL}}$

of the inference rules in Table 5 would be to set $s = \text{“expression”}$, and to let t , a , and e range over all syntactically correct expressions. This instantiation does not correspond to any known logic, but does provide some assurances that all variables are bound, the syntax is correct, and at least on a superficial level the constructed derivations are sensible.

3.3.1. Propositional Logic. As a trivial example, we can consider the inference rules parameterized by the abstract syntax subsets defined in Table 6 that only allow English phrase predicates that take no arguments to be atoms, and do not allow quantifiers in sentences. Under these conditions, the inference rules in Table 5 correspond precisely to a collection of axioms commonly associated with propositional logic (as presented in [8]). We can convince ourselves of this without much effort: the restriction imposed by the definition of e^{PL} ensures that the rules $[\forall\text{-INTRO}]$, $[\forall\text{-ELIM}]$, and $[\exists\text{-INTRO}]$ can never be employed in a valid derivation.

3.3.2. First Order Logic. A slightly more interesting example involves the abstract syntax subsets defined in Table 7.

TABLE 8. Abstract syntax subset for second order logic (SOL)

sorts	s^{SOL}	=	elem pred
terms	$t_{\text{elem}}^{\text{SOL}}$::=	x_{elem}
	$t_{\text{pred}}^{\text{SOL}}$::=	x_{pred} p
atoms	a^{SOL}	::=	$t_{\text{pred}}^{\text{SOL}}$ $t_{\text{pred}}^{\text{SOL}}(t_{\text{elem}}^{\text{SOL}}, \dots, t_{\text{elem}}^{\text{SOL}})$
sentences	e^{SOL}	::=	a^{SOL} $e_1^{\text{SOL}} \Rightarrow e_2^{\text{SOL}}$ $e_1^{\text{SOL}} \wedge e_2^{\text{SOL}}$ $e_1^{\text{SOL}} \vee e_2^{\text{SOL}}$
			$\neg e^{\text{SOL}}$ $\forall \bar{x}. e^{\text{SOL}}$ $\exists \bar{x}. e^{\text{SOL}}$

Theorem 3.2. *Let \mathcal{F} be the inference rules for first order logic. Then the inference rules in Table 5 are consistent relative to \mathcal{F} under $s^{\text{FOL}}, t^{\text{FOL}}, a^{\text{FOL}}, e^{\text{FOL}}$.*

Proof. The restriction imposed by the definition of e^{FOL} ensures that only sentences in first order logic can be inferred. The first premise of $[\forall\text{-INTRO}]$ ensures that all bound variables have a determined sort, while $[\forall\text{-ELIM}]$ and $[\exists\text{-INTRO}]$ ensure that any substitution respects sorts. \square

3.3.3. Second Order Logic. We can extend the quadruple in Table 7 by adding a sort for predicates to create an instantiation in Table 8 that corresponds to second order logic. The argument that the inference rules are consistent under this instantiation is analogous to the previous one.

3.4. Inference Rules for Statements

A document consists of a sequence of statements $s_1; \dots; s_n; \mathbf{end}$, where \mathbf{end} is implicit.⁴ 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 9. Using an **Assume** statement, a user can 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 5. In an introduction statement, the list of named variables is added to the variable context and any formulas dealing with those variables are removed from the assumption context. The function σ is extended to determine the sort of the variable in the same manner, but by

⁴There is no concrete syntax corresponding to \mathbf{end} , it is a placeholder representing an empty list of statements; it is inserted automatically to facilitate the recursive definition in Table 9.

TABLE 9. Inference rules for a list of proof statements.

$\text{[ASSUMPTION]} \frac{\Delta; \Phi \cup \{e\} \vdash S \quad FV(e) \subset \Delta}{\Delta, \Phi \vdash \text{Assume } e; S}$	$\text{[ASSERTION]} \frac{\Delta; \Phi \cup \{e\} \vdash S \quad FV(e) \subset \Delta \quad \Delta, \Phi \vdash e}{\Delta, \Phi \vdash \text{Assert } e; S}$
$\text{[INTRO]} \frac{\Delta, \bar{x} \mapsto \sigma(S, \bar{x}); \Phi \cap \{e \mid FV(e) \cap \bar{x} = \emptyset\} \vdash S}{\Delta, \Phi \vdash \text{Intro } \bar{x}; S}$	$\text{[BASE]} \frac{}{\Delta, \Phi \vdash \text{end}}$

examining all expressions found in the list of statements S that represents the rest of the document.

4. Verification, Search Algorithms, and Other Features

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 few new inference rules and a variety of other features. The new inference rules are treated as a recursive proof search algorithm that allows for a relatively simple characterization for search depth bounds, as described in Section 4.6.

We 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 $\Phi = \emptyset$ and $\Delta = \emptyset$, and then processes a list of statements as determined by the rules in Table 9, 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. 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 5. The only exception is the replacement of the [\wedge -INTRO] rule with the [\wedge -SEQ-INTRO] rule.

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

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

TABLE 10. Summary of basic verification algorithm.

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

More specifically, the algorithm always tries to apply the [CONTEXT] rule, and then uses pattern matching to check if any of the rules [IMPLIES-INTRO], [\wedge -SEQ-INTRO], [\vee -INTRO-L], [\vee -INTRO-R] or [\forall -INTRO] might apply (we call this “decomposition”). In these cases, premises are checked by using a recursive call to the verification algorithm.⁵ For the rules [IMPLIES-ELIM], [\wedge -ELIM-L], [\wedge -ELIM-R], [\vee -CASE], [NEGATION], [CONTRADICTION], [\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 10. It should be apparent that this algorithm can successfully verify any derivable formula, as long as each and every step of the derivation is presented to the algorithm in an appropriate, but not necessarily unique, sequence.

4.2. Basic Search Algorithm

In order to ease the reader into our presentation of the generalized search algorithm in Section 4.3 further below, we first define a basic search algorithm that does not involve substitutions. The fundamental goal of the search algorithm is to help verify an expression e by decomposing, instantiating, or combining the expressions within the context Φ under which e is verified. We believe that this algorithm can infer “proof steps” that are closer in complexity and size to those humans employ when doing formal reasoning. We demonstrate in Section 4.2.2 how this simple algorithm can easily generalize some of the inference rules in Table 5.

4.2.1. Basic Search Algorithm Definition. The basic search algorithm represented by the rules in Table 11 does not decompose the goal expression e that is being

⁵We discuss efficiency and termination in Section 4.6

TABLE 11. Inference rules for basic searching.

$\text{[BASIC-BASE]} \frac{}{\Delta; \Phi \vdash e \Rightarrow e}$
$\text{[BASIC-IMPLIES]} \frac{\Delta; \Phi \vdash e_2 \Rightarrow e \quad \Delta; \Phi \vdash e_1}{\Delta; \Phi \vdash (e_1 \Rightarrow e_2) \Rightarrow e}$
$\text{[BASIC-}\wedge\text{-L]} \frac{\Delta; \Phi \vdash e_1 \Rightarrow e}{\Delta; \Phi \vdash (e_1 \wedge e_2) \Rightarrow e} \quad \text{[BASIC-}\wedge\text{-R]} \frac{\Delta; \Phi \vdash e_2 \Rightarrow e}{\Delta; \Phi \vdash (e_1 \wedge e_2) \Rightarrow e}$
$\text{[V-CASE]} \frac{\Delta; \Phi \vdash e_1 \Rightarrow e \quad \Delta; \Phi \vdash e_2 \Rightarrow e}{\Delta; \Phi \vdash (e_1 \vee e_2) \Rightarrow e}$
$\text{[BASIC-NEGATION]} \frac{\Delta; \Phi \vdash e_0 \Rightarrow e}{\Delta; \Phi \vdash (\neg e_0) \Rightarrow e}$

verified, because this is the job of the basic verification algorithm. Rather, it attempts to decompose an expression that may imply the goal expression. In other words, it tries to verify that $\Delta, \Phi \vdash e_0 \Rightarrow e$ for some e_0 drawn from Φ .

The [V-CASE] rule is reproduced exactly from Table 5. It is included here because this rule fits well into the basic search algorithm, as we will demonstrate below. These rules are introduced into the verification algorithm by adding a new line in Table 10:

$\text{verify}(\Delta, \Phi, e) \quad := \quad \text{is } e \in \Phi?$
 $\quad \quad \quad \text{or, is } \Delta, \Phi \vdash e_0 \Rightarrow e \text{ derivable for some } e_0 \in \Phi?$
 $\quad \quad \quad \dots$

It is important to remember that the rules in Table 11 are presented *in addition* to the original inference rules in Table 5. The basic search algorithm represented by the rules in Table 11 is mutually recursive with the basic verification algorithm. When this algorithm needs to simply verify an individual expression (such as in the second premise of the [BASIC-IMPLIES] rule), it recursively calls the verification algorithm defined by $\text{verify}(\Delta, \Phi, e)$.

4.2.2. Algorithm Capabilities. The basic search algorithm makes many natural logical steps possible. For example, the following can be verified thanks to the fact that the [BASIC- \wedge -R] rule can be applied twice, followed by the [BASIC- \wedge -L] rule. It's important to note that the parentheses are only included for readability, and would be inserted automatically into the abstract syntax if they were not explicitly written in the concrete syntax.

Intro	$A, B, C, D, E.$
Assume	$A \wedge (B \wedge (C \wedge (D \wedge E))).$
Assert	$C.$

This capability is similar to what is done in the AproS system, the authors of which discuss search heuristics [15] in which an assertion is verified if it is a subexpression of an existing assumption. The algorithm provides a similar capability when dealing with disjunction. It is again worth noting that the parentheses are inserted for readability.

Intro	$A, B, C, D, E.$
Assume	$A \Rightarrow E.$
Assume	$B \Rightarrow E.$
Assume	$C \Rightarrow E.$
Assume	$D \Rightarrow E.$
Assume	$A \vee (B \vee (C \vee D)).$
Assert	$E.$

Here, the basic search algorithm will attempt to derive $\Delta, \Phi \vdash (A \vee (B \vee C \vee D)) \Rightarrow E$. By [V-CASE], this will lead to the premises $\Delta, \Phi \vdash A \Rightarrow E$ and $\Delta, \Phi \vdash (B \vee C \vee D) \Rightarrow E$. The former is already an assumption, and the latter will be decomposed once again by [V-CASE]. It should be clear how the rest of the verification occurs. This example illustrates that there is not always a need for a dedicated construct within the syntax for case distinction, such as the one found in the `Tutch` system [2]. Finally, the [BASIC-NEGATION] rule allows the verification process to take deeply-nested double negations into account, and the [BASIC-IMPLIES] rule, together with the [BASIC- \wedge -L] and [BASIC- \wedge -R] rules, enables the verification of Horn clauses.

4.3. Generalized Substitution Search Algorithm

4.3.1. Motivation. The generalized search algorithm extends the basic search algorithm by considering the possibility that \forall -elimination may need to take place before verification can occur. One way to think about this is to consider the [BASIC-BASE] rule in Table 11. One way to rewrite this rule is as follows:

$$[\text{BASIC-BASE2}] \frac{e_0 \equiv e}{\Delta; \Phi \vdash e_0 \Rightarrow e}$$

Suppose that e_0 contained some free variables, and we were allowed to substitute the free variables in e_0 in order to make it match e . If we call this substitution θ , we can rewrite our rule further as follows.

$$[\text{BASIC-BASE3}] \frac{\theta(e_0) \equiv e}{\Delta; \Phi \vdash \theta(e_0) \Rightarrow e}$$

It is also then natural to write the following rule, which motivates extending the rules in Table 11 to a generalized substitution search algorithm.

$$[\text{BASIC-BASE4}] \frac{\theta(e_0) \equiv e \quad \bar{x} \subset \text{dom}(\theta)}{\Delta; \Phi \vdash (\forall \bar{x}. e_0) \Rightarrow e}$$

TABLE 12. Inference rules for general searching.

[SEARCH-BASE] $\frac{\theta(e_0) \equiv e}{\Delta; \Phi \vdash \theta(e_0) \Rightarrow e}$	[SEARCH- \wedge -L] $\frac{\Delta; \Phi \vdash \theta(e_1) \Rightarrow e}{\Delta; \Phi \vdash \theta(e_1 \wedge e_2) \Rightarrow e}$	[SEARCH- \wedge -R] $\frac{\Delta; \Phi \vdash \theta(e_2) \Rightarrow e}{\Delta; \Phi \vdash \theta(e_1 \wedge e_2) \Rightarrow e}$
[SEARCH-IMPLIES] $\frac{\Delta; \Phi \vdash \theta(e_2) \Rightarrow e \quad \Delta; \Phi \vdash \theta(e_1)}{\Delta; \Phi \vdash \theta(e_1 \Rightarrow e_2) \Rightarrow e}$	[SEARCH- \forall] $\frac{\Delta; \Phi \vdash \theta(e_0) \Rightarrow e \quad \theta' = \theta - \bar{x}}{\Delta; \Phi \vdash \theta'(\forall \bar{x}. e_0) \Rightarrow e}$	

4.3.2. Definition. We 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 $\text{dom}(\theta) \cap \bar{x} = \emptyset$ and $FV(\text{ran}(\theta)) \cap \bar{x} = \emptyset$,

$$\theta(\forall \bar{x}. e) \equiv \forall \bar{x}. \theta(e).$$

Denote by $\theta' = \theta - \bar{x}$ the fact that

$$\begin{aligned} &\text{for all } y \in \text{dom}(\theta') \cap \text{dom}(\theta), \theta(y) \equiv \theta'(y), \\ &\text{that } \text{dom}(\theta') = \text{dom}(\theta) - \bar{x}, \\ &\text{that } \bar{x} \subset \text{dom}(\theta), \\ &\text{and that } FV(\text{ran}(\theta)) \cap \bar{x} = \emptyset. \end{aligned}$$

These equations, along with the inference rules for expressions in Table 5, can now be used to derive the generalized search rules in Table 12.

The basic verification algorithm can now be extended further. 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 $\text{dom}(\theta_0) = \emptyset$.

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

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

$$\begin{aligned} \text{verify}(\Delta, \Phi, e) &:= \text{is } e \in \Phi? \\ &\text{or, is } \Delta, \Phi \vdash e_0 \Rightarrow e \text{ derivable for some } e_0 \in \Phi? \\ &\text{or, is there } e' \in \Phi \text{ such that the [SEARCH] rule applies?} \\ &\dots \end{aligned}$$

The rules [SEARCH-BASE], [SEARCH- \wedge -L], [SEARCH- \wedge -R], [SEARCH-IMPLIES] in Table 12 are exactly like the corresponding rules in Table 11, with the exception of the introduced substitution. We can view the rules in Table 12 as a recursive algorithm that decomposes the left-hand side of the implication while maintaining a list of “substitutable” variables. The rule [SEARCH- \forall] introduces new variables

that can be substituted into this list. The [SEARCH-BASE] rule then takes advantage of this list in trying to form a match at the base cases of the algorithm.

It is again worth noting that the rules in 12 are not meant to be exclusive. For example, in the case of the second premise of the [SEARCH-IMPLIES] rule, it would be necessary to verify the premise $\Delta, \Phi \vdash \theta(e_1)$ by using the normal inference rules in Table 5 (in other words, it would be necessary to call `verify($\Delta, \Phi, \theta(e_1)$)`).

4.3.3. Limitations. In some cases, this search algorithm is not sufficient because it cannot construct a substitution that resolves all free variables on the left-hand side of an implication, as the unverifiable example below illustrates.

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

The search rules will try to find a substitution for the first assumption by matching the right-hand side of the implication under the quantifier with the target expression. Building the derivation from the bottom up, the [SEARCH- \forall] is encountered first in which

$$e_0 \equiv x < y \text{ and } y < z \text{ implies } x < z.$$

Next, the [SEARCH-IMPLIES] rule is required. At this point, the first premise of the [SEARCH-IMPLIES] rule requires that there be a θ such that $\theta(x < z) \Rightarrow 0 < 2$. Clearly, such a θ exists, mapping x to 0 and z to 2. However, there is a problem because the second premise requires that $\theta(x < y \text{ and } y < z)$ be verifiable. The variable y is not in the domain of θ , so this is impossible.

It is possible to address this limitation in *some* cases by applying the following rule (we use $\bar{x} \bar{y}$ to denote a single vector of variables which we can split into two parts \bar{x} and \bar{y}):

$$[\forall\text{-IMPLIES-LHS-PUSHDOWN}] \frac{\Delta; \Phi \vdash \forall \bar{x} \bar{y}. e_1 \Rightarrow e_2 \quad \bar{y} \cap FV(e_2) = \emptyset}{\Delta; \Phi \vdash \forall \bar{x}. (\exists \bar{y}. e_1) \Rightarrow e_2}$$

This rule allows us to push universally quantified variables that do not appear on the right-hand side of implications down under an existential quantifier on the left-hand side. In our 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 find that this adjustment is often sufficient for the instances of this situation that we encounter, as human authors seldom construct statements that preclude this transformation.

TABLE 13. Additional inference rules for existentially quantified expressions.

$[\exists\text{-SEQ-AND-INTRO}] \frac{\Delta; \Phi \vdash \exists \bar{x}. e_1 \quad \Delta, \bar{x}; (\Phi \cap \{e \mid FV(e) \cap \bar{x} = \emptyset\}) \cup \{e_1\} \vdash e_2}{\Delta; \Phi \vdash \exists \bar{x}. e_1 \wedge e_2}$
$[\exists\text{-SEQ-AND-ELIM-R}] \frac{\Delta; \Phi \vdash \exists \bar{x}. e_1 \wedge e_2}{\Delta; \Phi \vdash \exists \bar{x}. e_1}$

4.4. 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 13. 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.

4.5. Other Features

Our verifier includes a variety of additional features. Some of these could be included as a separate library of results written in the language. However, including them within the implementation improves performance and flexibility substantially. It is also consistent with our belief that the user practices enabled by these features deserve native support.

4.5.1. Evaluation Rules and Axioms Involving Constants. Our system implements over one hundred evaluation rules involving the constants in Table 4.

Mathematical functions and relations. Common operators, such as $\tilde{+}$, $\tilde{\max}$, $\tilde{\leq}$, $\tilde{[]}$, and so forth, evaluate as expected over rational constants.

Finite sets and tuples. Membership, union, intersection and product all work as expected on finite sets. Tuples can be concatenated using the overloaded \tilde{o} constant, and components of a tuple can be accessed using subscript notation (represented using the underscore symbol).

Quantification over finite sets. Given an expression of the form

$$\forall x_1, \dots, x_n. x_1 \tilde{\in} e_1, \dots, x_n \tilde{\in} e_n \Rightarrow e,$$

so long as the expressions e_i evaluate to finite sets, the expression e is evaluated under all possible assignments of x_1, \dots, x_n . If e evaluates to a boolean constant over all possibilities, the entire expression evaluates to the appropriate boolean constant. Otherwise, it does not evaluate at all. This also applies to instances of the existential quantifier.

This extensive support makes it possible to use our system as a sort of calculator. However, it is important to note that the system attempts to apply evaluation rules to all subexpressions when performing a verification, making the

verification process more flexible and powerful. Furthermore, a more modest collection of convenient axioms (about 50) are currently included in order to aid common practices. These primarily deal with the constants $\tilde{\mathbb{N}}$, $\tilde{\mathbb{Z}}$, $\tilde{\mathbb{Q}}$, and $\tilde{\mathbb{R}}$ and arithmetic operators. For example,

$$\frac{\otimes \in \{\tilde{+}, \tilde{-}, \tilde{\cdot}\} \quad \Delta, \Phi \vdash e_1 \tilde{\in} \tilde{\mathbb{Z}} \quad \Delta, \Phi \vdash e_2 \tilde{\in} \tilde{\mathbb{Z}}}{\Delta, \Phi \vdash e_1 \otimes e_2 \tilde{\in} \tilde{\mathbb{Z}}}$$

Having such rules built into the system makes it possible for the user to introduce axioms that limit quantified variables while still being able to substitute non-trivial arithmetic expressions for those variables. For example, if $\Delta, \Phi \vdash y \tilde{\in} \tilde{\mathbb{Z}}$ and $\Delta, \Phi \vdash z \tilde{\in} \tilde{\mathbb{Z}}$, the user can introduce an assumption of the form

$$\forall x. x \in \mathbb{Z} \Rightarrow \dots$$

and later substitute x with $y + z$ when she wishes to apply this assumption.

4.5.2. Support for Common Binary Relations. The use of binary relations is ubiquitous in mathematics, and these relations satisfy. Our system provides a few features to support the use of such relations. It is worth noting that while we specifically focus on constants presented in Table 4, any of these features can be extended to other constants by simply adding them to an appropriate list in our system's configuration files. It is fair to raise questions about whether it is too inflexible to always treat specific constants in this special manner. We argue that in the overwhelming majority of cases in which these symbols are used, they represent exactly these relations; furthermore, the use of these symbols for any relations for which these properties *do not* hold would constitute poor practice on the part of the user.

Equivalence classes of expressions. Equality is worth considering separately because a very common algebraic manipulation is the replacement of a component of an expression with a different but equivalent component. Thus, both the introduction of an equality and the verification of an equality are given special treatment.

Recall that the context Φ was treated as a simple set of expressions in the inference rules in Table 5 and Table 9 (such as in the premise of the rule [IMPLIES-INTRO]). It is possible to extend Φ to represent two sets: the set of expressions, and a set of equivalence classes of expressions. Let Φ_{eq} be this set of equivalence classes for a context Φ , and let $\Phi_{\text{eq}}(e)$ represent the equivalence class of an expression e (including a trivial class if e is not already found in Φ_{eq}) such that $e \in \Phi_{\text{eq}}(e)$. Then whenever a new expression of the form

$$e_1 \tilde{=} e_2$$

is introduced into Φ , the equivalence classes $\Phi_{\text{eq}}(e_1)$ and $\Phi_{\text{eq}}(e_2)$ are joined within Φ . Whenever the verification algorithm checks whether an expression of the form $e_1 \tilde{=} e_2$ is in Φ , it is checked whether $e_1 \in \Phi_{\text{eq}}(e_2)$ or $e_2 \in \Phi_{\text{eq}}(e_1)$. Furthermore, the syntactic equivalence function on expressions \equiv is augmented to check Φ_{eq} on every subexpression whenever it is called.

Binary relation graph. It is possible to further extend Φ by constructing a directed labelled graph in which nodes are the equivalence classes in Φ_{eq} and edges are labelled with one or more of the relation symbols $\{\tilde{<}, \tilde{>}, \tilde{\leq}, \tilde{\geq}, \tilde{=}, \tilde{\neq}\}$. Because most of the relations are transitive but one is not ($\tilde{=}$), the data structure supports both types of relations. Whenever an expression of the form $e_1 \otimes e_2$ is added to Φ in which $\otimes \in \{\tilde{<}, \tilde{>}, \tilde{\leq}, \tilde{\geq}, \tilde{=}, \tilde{\neq}\}$, an appropriate edge (or appropriate edges, since $\tilde{<}$ implies $\tilde{\leq}$) is added to the graph. Whenever the algorithm checks if an expression of this form is found in Φ , the graph is checked for an edge between the equivalence classes of the two sides of the inequality. If \otimes is a transitive relation, the graph is also checked for a path between the two equivalence classes.

Algebraic Manipulations. Many algebraic manipulations can be supported by defining various normal forms for expressions involving commutative and associative operators. The operators $\{\tilde{+}, \tilde{-}, \tilde{\cdot}, \tilde{\div}, \tilde{\min}, \tilde{\max}\}$ are both commutative and associative, and so, are transformed into folds over sets (e.g. $(1\tilde{+}(2\tilde{+}3))$ and $((1\tilde{+}2)\tilde{+}3)$ both become $\sum_{x \in \{1,2,3\}} x$). The subtraction operator ($\tilde{-}$) is replaced with addition and the right-hand argument is negated. Furthermore, equations and inequalities are normalized. For example, the inequality

$$a\tilde{+}b \tilde{<} y\tilde{-}z$$

is conceptually transformed into

$$\left(\sum_{x \in \{a,b,-y,z\}} x \right) = 0.$$

Evaluation of Simple Functions. One-step evaluation of functions defined within the context using the existing syntax can be supported in an ad hoc manner by extending the evaluation algorithm for constants. The algorithm takes the assumption context Φ as a parameter. For any subexpression it considers that is of the form fe where f is some bound variable and e is an expression (possibly a tuple), it searches the context for a definition of the form

$$\forall \bar{x}. f' e_1 = e_2$$

such that $f, f' \notin \bar{x}$, such that $\text{match}(\bar{x}, f' e_1, fe)$ succeeds, and such that e_2 evaluates to a constant under the substitution generated by this match. If it succeeds in finding such an assumption, it replaces the subexpression fe with the resulting constant. To more concretely illustrate the feature, we provide the following verifiable example.

Intro	f .
Assume	for any x , $f(x) = x + x$.
Assert	$f(1) = 2$.

While this is a very limited feature, it suggests an approach for introducing more powerful features *without* extending the concrete syntax or adding any additional specialized structures to the abstract syntax.

4.6. 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

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

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

Several similar systems exist whose development was motivated by observations and goals similar to ours. Our system can be viewed in part as further development and integration of the various ideas incorporated into those systems, and furthermore as an example of which priorities and features of those systems we believe should be emphasized (and which are disadvantageous and may need to be removed or modified) if we hope to improve the practical usefulness and accessibility of verification systems.

Our system’s syntax of statements shares the simple and austere structure of the `Tutch` system [2], but introduces a more rich (but still familiar to most intended users) syntax for expressions. Our system’s verification algorithm is similarly motivated by the need to extend verifiability beyond the basic logical steps (applications of an inference rule) to more human-friendly proof steps in which multiple rules are applied implicitly. Our system provides native support for verifying chains of algebraic manipulations on equations, and this feature is recognized by the authors of the `Tutch` system as essential to making further progress.

We have already explicitly noted a variety of ways in which our system is similar to `Scunak` [3]. A few important differences is our approach to types. We strongly believe that types and type checking are *not* as universal as many other concepts and activities, such as sets and algebraic manipulations, and so our system does not support types natively. However, users of our system are free to use

sets as if they were types wherever they see fit to do so. In fact, the designers of Scunak chose a *simpler* type system than the sophisticated type systems of Coq and Twelf, and we believe this is not merely a temporary measure, but at least an implicit recognition that such sophisticated type systems do not correspond to the well-established intuitions of users. Finally, it is worth noting that we disagree with the designers of Scunak that proofs of fundamental concepts, such as algebraic distribution laws, are a good touchstone for measuring the success of proof verification systems. On the contrary, focusing on such proofs distracts from efforts directed towards recognizing the larger proof steps humans prefer to use in practice, ones in which distributivity is one of many natural, implicit manipulations.

The design of the EPGY Theorem-Proving Environment [10] fulfills many of our stated goals. Especially noteworthy is the extensive support it provides for algebraic manipulations and computations. However, the environment is fundamentally a teaching tool and is not sufficiently flexible for formalizing novel research results. Most importantly, the lightweight approach we advocate is not directly supported by the system. The designers of the ForTheL language and SAD proof assistant [18] share our motivations and suggest similar principles. Many arguments written using the ForTheL syntax look like natural language text, and the design of the language reflects common high-level constructs used in formal arguments. However, the text uses a few unfamiliar conventions and deviates from the notation employed for mathematical formulas in \LaTeX . Furthermore, the high-level constructs for managing proofs are reminiscent of proof scripts for interactive theorem provers and present similar obstacles to new users. Enforcing high-level constructs like proofs and theorems inhibits the user from employing a lightweight approach.

In recent work [4], the Ω MEGA proof verifier [16] has been integrated with the scientific text editor $\text{\TeX}_{\text{MACS}}$. 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. Another 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.

More widely, there exist other efforts to create interfaces and systems for practical formalization of mathematics. The MathLang project [7] 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 [20]. Some of these tools provide a way to construct proofs by induction, such as Coq [12], PVS [11], and Isabelle [13, 14]. More specifically, formal representation and verification systems include Isabelle/Isar [19] and Mizar [17]. 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 [6]. 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 lightweight verifier can be augmented with a variety of heuristic search strategies that further enhance usability.

There are many obvious avenues for further work. We are interested in organizing the system in a way that allows the instantiation of the inference rule template not just with syntactic restrictions but also specific inference rules. For example, it should be possible to tell the user that she has successfully restricted her work to the axioms of Presburger arithmetic or intuitionistic logic. This requires the system to relate heuristic verification rules involving particular constants with particular logical systems. While we do not believe that the basic structure of the representation should be augmented much further, it should be possible to allow the user to define new constants using a new kind of statement. We also have high hopes for automated inference of user intention directed by heuristic syntactic patterns. For example, it should be possible to detect collections of axioms within the existing syntax that correspond to pure, finitely computable recursive functions as well as to basic type systems (such as the type system of the simply-typed lambda calculus). Such definitions should be recognized as such by the verifier so that it can use the guarantee of termination they provide to do more exhaustive searches for derivations and results.

Our work so far has focused on supporting forward reasoning. Of course, a user is free to write a proof in any order, but it is worth investigating whether the existing axioms for disjunction are sufficient for the sort of goal-directed reasoning that many systems that focus on induction, such as Coq, support directly. 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. It is our hope that these techniques can eventually lead to tools that can readily be used by researchers working on novel results. For the verification algorithm, a variety of additional verification and evaluation strategies are planned to support specific domains (such as graph theory and algorithms), and we hope that it will eventually be possible to work within these domains by using the current language idiomatically, with minimal extensions to the syntax.

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] A. Abel, B. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic, 2001.
- [3] C. E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *Mathematical Knowledge Management, MKM 2006*, page 110, Wokingham, England.

- [4] 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 Calculemus symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 398–414, Berlin, Heidelberg, 2008. Springer-Verlag.
- [5] 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.
- [6] D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.
- [7] F. Kamareddine and J. B. Wells. Computerizing Mathematical Text with MathLang. *Electron. Notes Theor. Comput. Sci.*, 205:5–30, 2008.
- [8] S. C. Kleene. *Mathematical Logic*. Dover Publications, 1967.
- [9] D. Leijen and E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, Departement of Computer Science, Universiteit Utrecht, 2001.
- [10] D. McMath, M. Rozenfeld, and R. Sommer. A Computer Environment for Writing Ordinary Mathematical Proofs. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 507–516, London, UK, 2001. Springer-Verlag.
- [11] 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.
- [12] C. Parent-Vigouroux. Verifying programs in the calculus of inductive constructions. *Formal Aspects of Computing*, 9(5-6):484–517, 1997.
- [13] L. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994.
- [14] L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and Its Applications*. MIT Press, 1997.
- [15] W. Sieg and S. Cittadini. Normal Natural Deduction Proofs (in Non-classical Logics). In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005.
- [16] 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.
- [17] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In *Proc. of the 9th IJCAI*, pages 26–28, Los Angeles, CA, 1985.
- [18] K. Verchinine, A. Lyaletski, A. Paskevich, and A. Anisimov. On Correctness of Mathematical Texts from a Logical and Practical Point of View. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 583–598, Berlin, Heidelberg, 2008. Springer-Verlag.
- [19] 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.

- [20] 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