Ontology Support for a Lightweight Formal Verification System*

Andrei Lapets Prakash Lalwani Assaf Kfoury

Abstract

The usability of verification systems is becoming increasingly important, and the effective integration of ontologies of formal facts (definitions, propositions, and syntactic idioms) into machine verification systems will likely play a role in improving the usability of such systems. The AARTIFACT lightweight verification system utilizes an ontology of formal propositions in order to support lightweight verification of formal arguments that involve common mathematical concepts. The ontology is stored within a relational database, and can be assembled and extended using a simple web interface by contributors who are domain experts. The database can be compiled into two separate components of the AARTIFACT system: a verifier component that computes congruence closures of expressions containing relations and predicates found in the ontology, and a JavaScript application that interactively presents to users information about the constants, operators, relations, predicates, syntactic constructs, and idioms found in the ontology (and, thus, supported by the verifier). In this way, the database serves to improve both the verification system's capacity to infer implicit applications of logical propositions within a user's formal argument, and to inform users in a context-aware and structured manner of the verification system's capabilities and limitations.

1 Introduction

Machine verification of formal arguments can increase our confidence in the correctness of those arguments, and can provide benefits ranging from unbound variable detection to easy dissemination of fully certified formal proofs. However, the costs of employing machine verification still outweigh the benefits for a variety of formal reasoning activities, so features that contribute significantly to various aspects of a verification system's usability are becoming increasingly important in the design of machine verification tools [1].

One promising direction of work involves the integration of large databases (or ontologies) of existing formal facts (definitions, propositions, syntactic idioms) into machine verification systems. This approach has been taken in some domain-specific contexts and is closely related to a wide variety of existing work in finding ways to assemble and expose to the user libraries of formal results [2, 7, 11, 22]. This approach has several benefits. When a library of existing propositions and syntactic idioms is available, a machine verification system is more likely able to meet the expectations of users who are already familiar with the formal constructs and manipulations they wish to employ in their arguments. Furthermore, it is easier for expert users to make a trade-off between flexibility and confidence in verification by introducing their own high-level concepts and facts without having to represent them using low-level logical constructs.

The AARTIFACT lightweight verification system[1] is designed to support the authoring and validation of formal arguments involving basic, ubiquitous mathematical concepts (e.g. numbers, sets, vectors, graphs). The system serves as a prototype for investigating potential techniques for improving the usability of formal verification systems. It is designed around a database of simple formal propositions that deal with common mathematical concepts, properties, and relationships, as well as with syntactic idioms that represent these. A web interface to browse the ontology is made available to the user (an approach

*This material is based upon work partially supported by the National Science Foundation under Grant No. 0820138 and Grant No. 0720604.

[1] Interactive demonstration available at `http://www.aartifact.org`.

used by others in earlier related work [7]), but the ontology is also compiled into two important system components that provide interactive aid in the authoring and verification of formal arguments.

This paper discusses the design, implementation, and utilization of the ontology underlying the AARTIFACT system's verification capabilities. Related papers discuss the underlying motivation and structure of the overall verification system [16], as well as the design of its user interface [19]. The rest of this paper is organized as follows. Section 2 describes at a high level the overall design and operation of the AARTIFACT system; in particular, the role of the ontology and its relationship to other components is described. Section 3 discusses the definition and implementation of the ontology. Section 4 illustrates through examples some benefits of an integrated ontology of common concepts in the AARTIFACT verification system. Section 5 discusses related work and concludes.

2 Overview of System Components and Operation

Figure 1 illustrates the components of the AARTIFACT system. An expert-managed ontology of propositions is maintained within a relational database. These expert managers can add logical formulas to the database that are of a very specific form (described in Section 3) which is amenable to the inference capabilities of the system's validation procedure [16]. The contents of the database are compiled into two separate applications. One application is a server-side verification application that can be run when requests are submitted to it. Using a web interface, the user can submit an argument for verification using the click of a button and receive immediate validation feedback from the server. The other application is a client-side JavaScript module that automatically informs the user, as they author a formal argument, of syntactic idioms available in the ontology based on the text surrounding the location of the user's cursor.

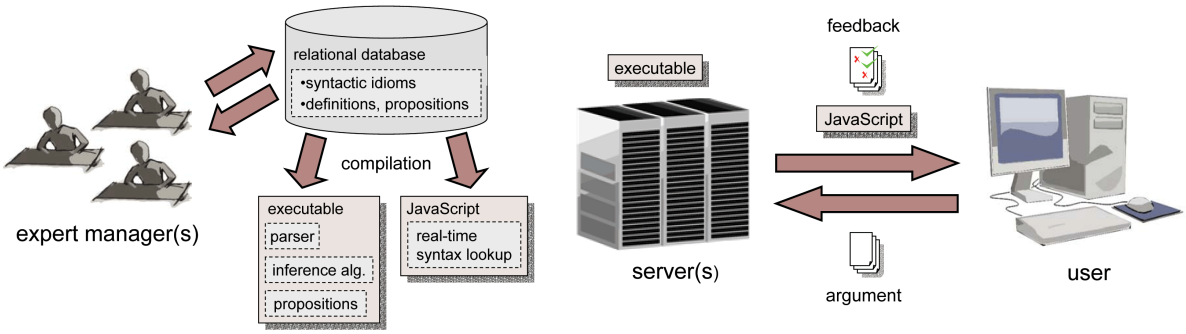

Figure 1: Overview of system components (left) and operation (right).

The representation of formal facts is an important consideration in the design of a formal ontology. In particular, it is important to adopt representations that are amenable to translation and compilation procedures that can convert the ontology's contents into other useful representations. For example, the contents of the ontology can be included in a JavaScript application that provides interactive hints for supported syntactic constructs and idioms. It can also be used for constructing parsers, as well as for constructing algorithms that perform closure computations. This suggests that there may be opportunities for applying existing knowledge and experience from work in compiler design in the construction of machine verification systems.

The ontology integrated into the AARTIFACT system supports two important capabilities that contribute to usability: implicit reasoning and use of syntactic idioms.

Implicit Reasoning with Common Mathematical Concepts. Even if one considers a small collection of mathematical concepts, a practicing mathematician is familiar with a large number and a great variety of propositions that describe relationships between the concepts in this collection. While engaging in formal reasoning, a user can utilize this knowledge without much conscious effort. A formal reasoning system will act as a hindrance if it requires that the user consciously recognize and explicitly state her application of this knowledge. As the designers of the `Tutch` system posit, an "explicit reference [to an inference rule] to [humans] interrupts rather than supports the flow of reasoning" [1]. The designers of the Scunak system [5] refer to a system's capability to recognize implicit application of such knowledge as "[retrievability] ... by content rather than by name."

The AARTIFACT verification system supports validation of *implicit* invocations within a formal argument of any of the propositions within the database. This is accomplished by maintaining a data structure during the verification process that computes congruence closures [3] of expressions that contain any of the relations found in the database. This data structure and the closure computation are described in a relevant report [17].

Syntactic Idioms and Real-time Lookup. Syntax is a means of communication, and a simple and natural formal syntax is advantageous to a verification system because it provides a method that can be learned quickly for encoding formal arguments [1, 5, 27]. However, this simple syntax must then be used to represent arguments that can contain references to a large library of operators, predicates, and even syntactic idioms. It is necessary to maintain all these in a database in order to support conversion of these idioms into some normal form, and to expose them to users without requiring that they spend time and effort browsing the database. While an indexed database of syntactic idioms and logical propositions is a natural starting point, real-time keyword-based lookup techniques for programming environments [20] suggest a means for further improving usability. While using the AARTIFACT system, the user is able to learn relevant constructs supported by the system (and any limitations that might exist) while authoring arguments. This is accomplished by delivering the JavaScript application compiled from the database to the user's browser. As the user is typing, the application maps keywords to syntaxes and examples derived from the contents of the database.

3 Ontology Design and Implementation

The AARTIFACT ontology is designed for maintaining a collection of propositions and definitions dealing with common mathematical concepts (numbers, sets, vectors), properties they may have (the sums of their elements, the subsets and subvectors they may have and their properties, etc.), and relationships that may hold between them. The following proposition represents a very simple example:

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

Propositions such as the above deal with definitions, properties, and relationships that govern concepts. However, it is also possible to maintain propositions that state an equivalence between two forms of notation or syntax. These can be viewed as establishing a normal form for representing certain concepts or properties thereof. For example, the following proposition converts the typical notation for a sum of a finite range of components in a vector, "$v_i + \ldots + v_j$", into a predicate that is then used in other propositions about the properties of this concept:

" for any v, i, j,
v is a vector , $v \in \mathbb{Z}^{|v|}$, $0 \leq i$, $i \leq j$, $j < |v|$,
implies that
$v_i + \ldots + v_j$ is the sum of components in v in index range i to j".

Database Maintenance. There exists a simple web form that allows a domain expert (or a group of experts) to submit new entries or manage the existing entries within the ontology. It also allows users to browse and search for formulas by the constants, operators, and predicates they contain using the interactive, dynamically-generated HTML interface presented in Figure 2. While a formula is implicitly associated with particular disciplines in part by the constants, operators, and predicates that it contains, the database includes a simplistic tagging mechanism to better accommodate categorization of formulas. In particular, it is possible to label a formula with the logical system(s) with respect to which the formula is sound. While the consistency of the overall database is never checked or maintained, this capability allows portions of the database that are consistent with a particular logic to be assembled and to be employed exclusively by the validation procedure.

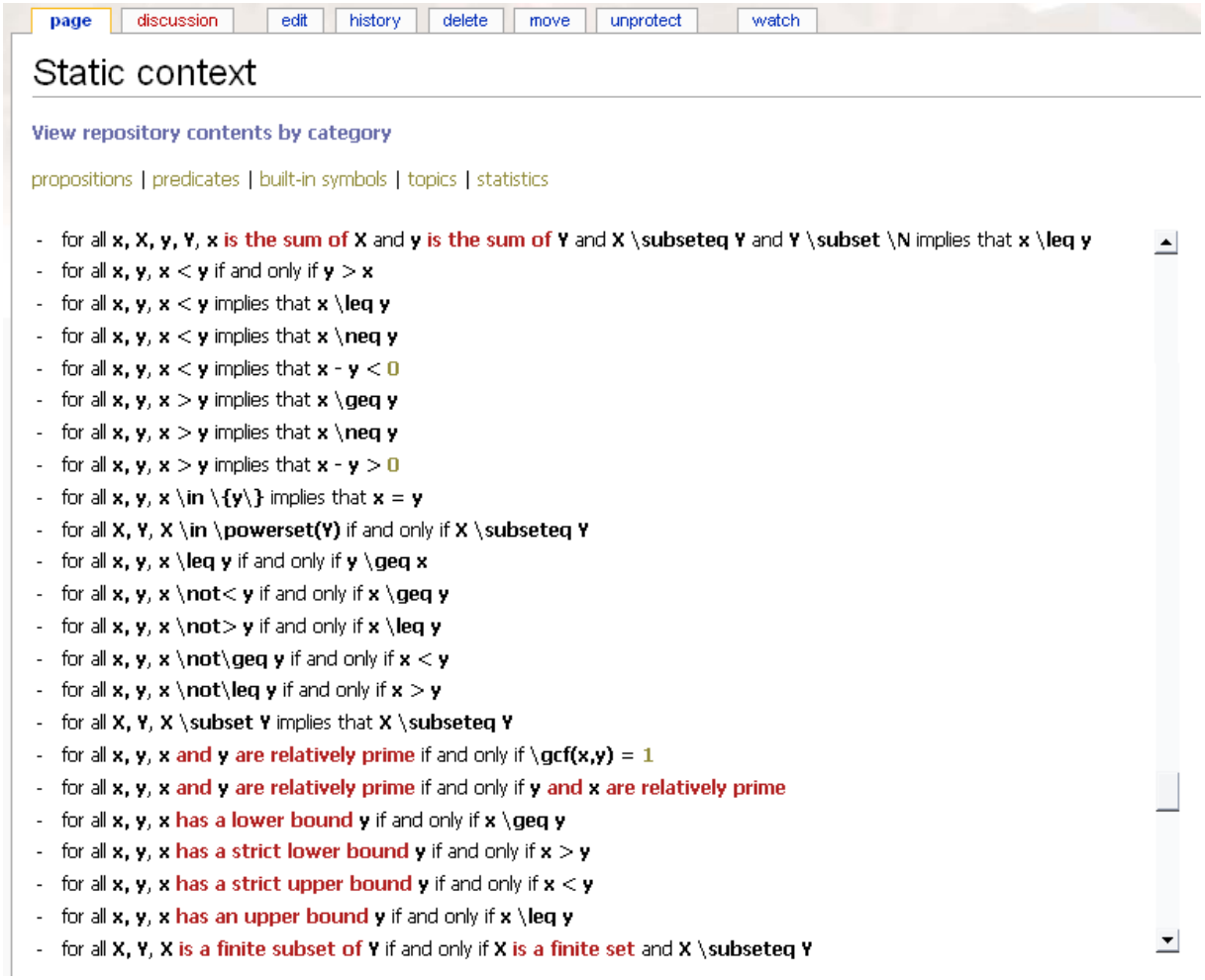

Figure 2: Screen capture of the interface for browsing propositions within the ontology.

Integration with Verification. The AARTIFACT system's inference algorithm relies on a collection of inference rules corresponding to those found in a typical definition of higher-order logic [13] (i.e. those governing conjunction, disjunction, negation, and quantification), and variants thereof found in common sequent calculus formulations [9]. The logical inference rules of the system and its validation procedure are presented in more detail in a relevant report [14]. The validation procedure verifies arguments with respect to these inference rules as well as the entire database of propositions. It is similar to validation procedures in related work, such as the proof checking algorithm of the `Tutch` system [1]. The distinguishing feature of our procedure is the more sophisticated assumption context, which incorporates the entire ontology and computes closures against it. This is described in more detail in relevant papers and reports [14, 16, 17].

The verification capabilities provided by the AARTIFACT system are "lightweight" in that it is not essential that any guarantee be provided about the logical consistency or completeness of the validation process (allowing a user to validate that, for example, the proper syntax is used, that all variables are bound, or that only "intuitive" symbolic manipulations are employed). However, the soundness of the validation process can be guaranteed if the system can use only a subset of the inference rules and database propositions that is consistent with a particular logic. This capability has been demonstrated for propositional and first-order logic.

Form and Representation of Ontology Propositions. All propositions within the database are of the form

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

where $\bar{x}$ represents a list of variables, $r_1, \ldots, r_{n+1}$ are common relations (such as $\leq$ as well as English predicate such as "X `is a set`"), and the entries $\bar{u}_1, \ldots, \bar{u}_{n+1}$ are either constants (such as "2", "$\mathbb{R}$", or "∞") or variables drawn from the list $\bar{x}$. The second example presented above cannot be represented directly in this form, so a simple recursive compilation procedure is utilized to convert such formulas. For example, given an expression of the form "$x + y > z$", it is possible to introduce a fresh variable v and produce two new expressions: "v is the sum of x and y" and "$v > z$". In this way, an expression involving any nested combination of operators can be converted into the necessary form.

Database Schema and Management. When a designer or expert manager submits a formula using the web form presented in Figure 2, the concrete syntax is parsed and converted into an abstract syntax tree using the same parsing mechanism employed by the verifier's user interface. The abstract syntax tree consists of expression nodes, and a compilation process compiles this tree into a flat list of node descriptions. Each description includes an identifying node index, the index of the node's parent node (if any), and the index range that includes all the indices of the children of the node. It also includes indices and markers that can be used to determine the individual expression node's syntactic role (which can be used to produce a formatted user-friendly HTML string of the entire formal expression) or semantic content (which can be used to, for example, index the entire formal expression using the constants found within it). For example, a boolean marker denotes whether a node is necessary only for displaying a user-friendly expression. This list of descriptions is then inserted into a flat table within the relational database. This table has sufficient information to reconstruct any expression because each node description specifies the range of child indices, and the order of the indices of the child node descriptions corresponds to the order of the child nodes within the abstract syntax tree.

Whenever the executable component of the AARTIFACT system that performs verification of user arguments is compiled, the database is converted into an initial instance of the dynamic context and incorporated into the executable component. The definition, implementation, and performance of the dynamic context and its associated closure operation are described in an earlier report [17].

4 Benefits of Ontology-supported Lightweight Verification

We have utilized [18] the AARTIFACT system in defining and reasoning about a compositional formalism for a typed domain-specific language [4]. The ontology propositions dealing with the algebra of sets were essential in making this process manageable and in allowing the proofs to be legible. This exercise also led to the discovery of a few minor errors and to the simplification of a few definitions. The system has also been deployed within two undergraduate courses: an advanced undergraduate course on functional programming (described in a related report) [15] and an introductory undergraduate course in linear algebra (described in a related paper [19]). These experiences demonstrated the usability benefits of meeting student expectations with the help of an ontology of relationships between common concepts.

Assume for any C, T, (C,T) is a service-level agreement iff $C \in \mathbb{N}$, $T \in \mathbb{N}$, $C \geq 1$, and $T \geq 1$.

Assume for any v, C, T,
    v satisfies (C,T)
iff
    (C,T) is a service-level agreement,
    $v \in \{0,1\}^{\infty}$, and for all $m \in \mathbb{N}$, $v_m + ... + v_{m+(T-1)} \geq C$.

Assert for any v, C, T,
    if (C,T) is a service-level agreement and v satisfies (C,T) then
        for all $T' \in \mathbb{N}$,
            if $T' \geq T$ then
                (C,T') is a service-level agreement, and
                for all $m \in \mathbb{N}$,
                    $v_m + ... + v_{m+(T-1)} \geq C$,
                    $v_m + ... + v_{m+(T'-1)} \geq v_m + ... + v_{m+(T-1)}$,
                    $v_m + ... + v_{m+(T'-1)} \geq C$,
                for all $m \in \mathbb{N}$, $v_m + ... + v_{m+(T'-1)} \geq C$,
                v satisfies (C,T').

Figure 3: A verifiable formal argument about the safety of a simple service-level agreement transformation. The argument makes implicit use of properties of integers, indexed ranges of vectors, summations thereof, and other concepts.

4.1 Reasoning with Common Mathematical Concepts

In this section, we consider a use case involving a particular scheduling formalism [12] for modelling safe transformations on service-level agreements. In order to avoid introducing more sophisticated concepts specific to the field of research dealing with scheduling, we express the examples in abstract terms that deal only with concepts such as sums and vectors. However, it should be noted that the formal argument in this abstract example is pertinent to the safety of scheduling algorithms that rely on the scheduling formalism's safety [12].

In the scheduling formalism we consider in this example, there are resources and there are tasks that require resources. For example, a resource might be a processor and tasks might be operating system

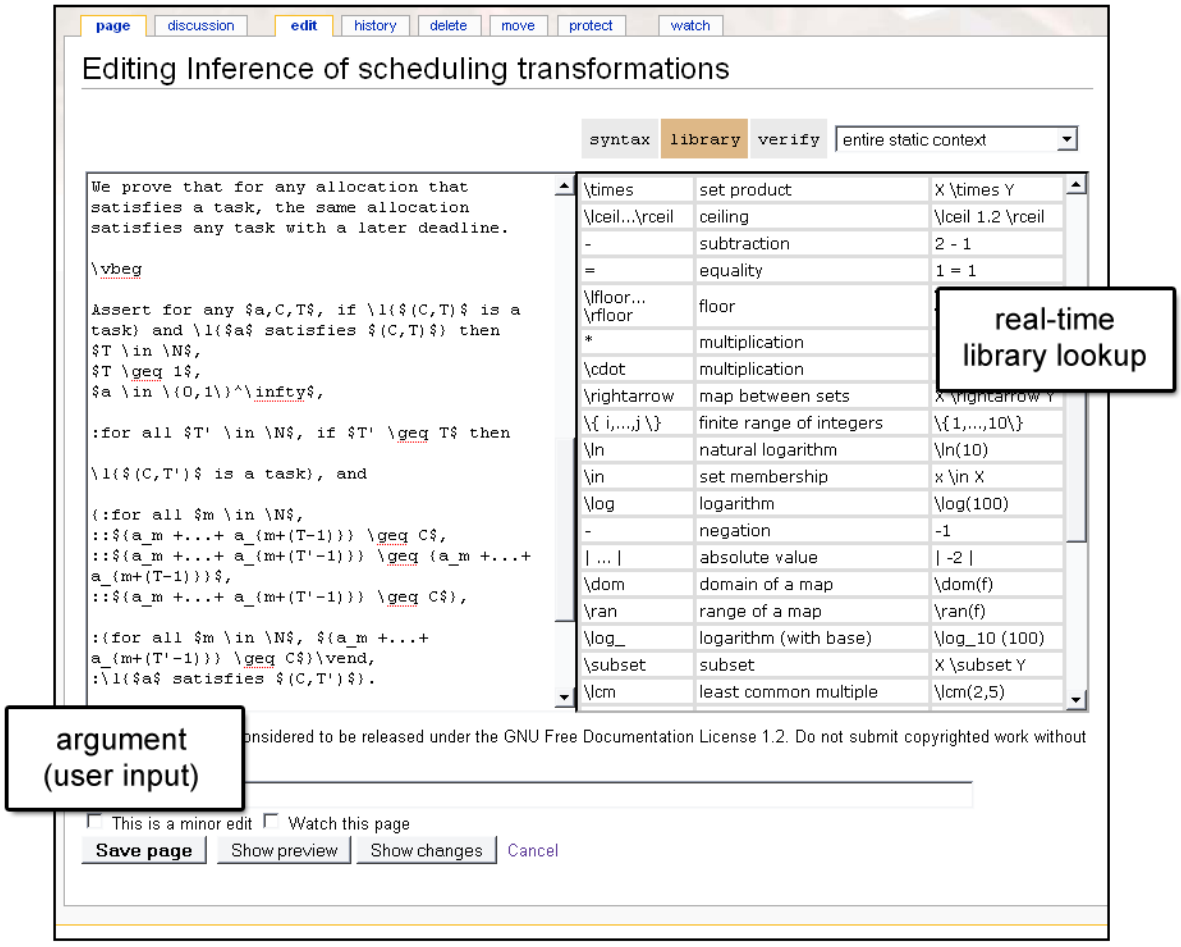

Figure 4: Screen capture of the user interface in "library" mode.

processes. Given a task, a resource is either allocated or not allocated to that task at each discrete time step according to a *schedule*, which can be represented as an infinite vector (i.e. sequence) of binary digits $v \in \{0,1\}^\infty$. If the nth entry in this vector is 1, this indicates that the resource is allocated to the task at that time step. If it is 0, then the resource is not allocated at that time step. The total amount of time a schedule v allocates a resource to a task over some span of time from time step i to time step j is the sum of the vector components in that range, $v_i + \ldots + v_j$. A *service-level agreement* is a pair of natural numbers (C,T) that represents a requirement that a schedule might satisfy: that every span (i.e. subvector) of T contiguous time steps has at least C time units allocated to the resource (that is, at least C non-zero entries).

Theorems about this formalism require the application of intuitive facts about the relationships between the lengths and sums of subvectors of a vector $v \in \{0,1\}^\infty$. We consider the following example of such a fact. Within in a vector v, if all subvectors of length at least n have sum at least k, then any subvector of v having length $n \cdot m$ must necessarily have a sum of at least $k \cdot m$. This intuitive result requires little effort for the user to verify informally. An effective formal verification system should be able to infer this fact about a vector given its stated properties.

In order to allow users to use this result implicitly in their arguments, the following proposition can be introduced into an ontology meant to support reasoning about the formalism we are considering:

" for any $u, v, w, i, j, i', j', s, t$
u is a vector , $i' \leq i$, $j \leq j'$,
v is a subvector of u in the index range i to j,
w is a subvector of u in the index range i' to j',
s is the sum of v,
t is the sum of w,
implies that
$t \geq s$".

This ensures that implicit applications of this fact can be verified by the validation procedure of the verification system. Figure 3 illustrates a verifiable formal argument that implicitly applies the propositions presented in this section. This argument defines explicitly the notion of a service-level agreement, as well as what is means for a schedule to satisfy an agreement. Then, it provides a verifiable assertion that for any schedule v and agreement (C,T), if v satisfies (C,T) then it also satisfies (C,T') for any $T' \geq T$. That is, widening the span of time steps in question preserves the conditions for satisfaction.

4.2 Interactive Lookup of Ontology Contents

Figure 4 illustrates the AARTIFACT system's verification web interface from the user's perspective. The user submits a formal *argument* represented using concrete syntax. If the "library" or "syntax" tab is selected, real-time hints for supported constants, relations, and syntax from the ontology are provided interactively based on the text surrounding the cursor as the user types. This is accomplished by accumulating the constants, relations, and predicates found in the propositions in the ontology and then converting them into a JavaScript array that contains each of these possible entries along with a collection of index keywords that correspond to it. These index keywords can also be added manually to the ontology by domain experts.

This feature allows the user to interactively explore the contents of the ontology with little effort. Because the ontology may contain a variety of predicates and constants whose exact forms might not be familiar to the user, this feature is essential for making utilization of the ontology's contents manageable.

5 Related and Future Work

One of the purposes of the ontology utilized by the AARTIFACT system is to support user-friendly syntactic constructs and idioms. This capability is recognized as important within several efforts and projects aiming to design and implement systems with similar usability characteristics [1, 5, 10, 21, 24, 25, 27].

More generally, our work is concerned with the assembly of an ontology of abstract knowledge about a formal domain. There is a particular emphasis in our work on collecting high-level, common-sense facts about formal concepts that are frequently utilized. In taking this approach, we are inspired by work in knowledge assembly from subdisciplines of artificial intelligence. Examples include the Cyc Project [23] and the Open Mind Common Sense project [26, 8]. The aim of these projects is to collect a large database of commonsense propositions about the real world from users. However, the collected data in these projects need not focus on particular formal domains. Work also exists in assembling indexed databases of *computational* methods (algorithms) that can be utilized through an interface. One example is the online search tool Hoogle, designed to allow users to explore the Haskell libraries [22]. Another example with the same underlying motivation is the WolframAlpha "answer engine" (as opposed to *search* engine) [28], which allows users to interact through a natural language interface with a collection

of databases that contain both facts and algorithms. However, natural language is not necessarily an effective interface for a database of propositions or computational methods if it does not provide a means for specifying the context of a query. In particular, even a human capable of communicating in a natural language cannot answer queries when the queries are posed out of context.

Matita [2] is a proof assistant the automation of which is heavily based on an integrated search engine for an underlying library. While the approach in our work is similar, our work focuses more on facilities for maintaining a "lightweight" library that includes propositions governing intuitive, high-level relationships between concepts.

The dynamic context that allows utilization of the ontology and its corresponding closure operation are very similar to the data structures and algorithms found in work on congruence closures [3]. The contribution of this work is to apply this technique in conjunction with a database of purely symbolic propositions involving constructs that can correspond to highly abstract but still common mathematical concepts. That is, the congruence closure is computed with respect to a *large* collection of both low-level (e.g. "$<$") and high-level (e.g. "`is a perfect matching`") relations.

Further extensions to the ontology can be considered. In particular, it should be possible to extend support to slightly more complex propositions, such as those involving at least one existential quantifier, or even those containing higher-order predicates. It may also be worthwhile to introduce input and output support for standards such as MathML [6]. The interface for managing and exploring supported propositions can be extended with additional analysis capabilities. In particular, it would be useful to be able to analyze, or even visualize, relationships and dependencies between propositions. The ability to search for formulas using structural similarity, as well as to automatically index relationships between analogous formulas, would also be of value.

References

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

[2] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

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

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

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

[6] Stephen Buswell, Stan Devitt, Angel Diaz, Patrick Ion, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) 1.01 Specification (Abstract). Retrieved June 21, 2010., 1999.

[7] Paul Cairns and Jeremy Gow. Integrating Searching and Authoring in Mizar. *Journal of Automated Reasoning*, 39(2):141–160, 2007.

[8] Timothy Chklovski and Yolanda Gil. Improving the design of intelligent acquisition interfaces for collecting world knowledge from web contributors. In *K-CAP '05: Proceedings of the 3rd international conference on Knowledge capture*, pages 35–42, New York, NY, USA, 2005. ACM.

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

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

[11] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Vatche Ishakian, Azer Bestavros, and Assaf Kfoury. A Type-Theoretic Framework for Efficient and Safe Colocation of Periodic Real-time Systems. In *RTCSA '10: The 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Macau, China, August 2010 Forthcoming. IEEE Computer Society.

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

[14] Andrei Lapets. Improving the accessibility of lightweight formal verification systems. Technical Report BUCS-TR-2009-015, CS Dept., Boston University, April 30 2009.

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

[16] Andrei Lapets. User-friendly Support for Common Concepts in a Lightweight Verifier. In *Proceedings of VERIFY-2010: The 6th International Verification Workshop*, Edinburgh, UK, July 2010.

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

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

[19] Andrei Lapets and Assaf Kfoury. A User-friendly Interface for a Lightweight Verification System. In *Proceedings of UITP'10: 9th International Workshop On User Interfaces for Theorem Provers*, Edinburgh, UK, July 2010.

[20] Greg Little and Robert C. Miller. Keyword programming in java. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 84–93, New York, NY, USA, 2007. ACM.

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

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

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

[24] Wilfried Sieg and Saverio Cittadini. Normal natural deduction proofs (in non-classical logics). In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005.

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

[26] P. Singh, B. Barry, and H. Liu. Teaching machines about everyday life. *BT Technology Journal*, 22(4):227–240, 2004.

[27] Konstantin Verchinine, Alexander Lyaletski, Andrei Paskevich, and Anatoly Anisimov. On Correctness of Mathematical Texts from a Logical and Practical Point of View. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 583–598, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Wolfram Alpha L.L.C. WolframAlpha computational knowledge engine. Available at: http://www.wolframalpha.com/about.html.