Meta-Logical Frameworks and Formal Digital Libraries [⋆]

Carsten Schürmann

Department of Computer Science
Yale University
carsten@cs.yale.edu

Although the Annals will publish Dr. Hales's paper, Peter Sarnak, an editor of the Annals, whose own work does not involve the use of computers, says that the paper will be accompanied by an unusual disclaimer, stating that the computer programs accompanying the paper have not undergone peer review. There is a simple reason for that, Dr. Sarnak says it is impossible to find peers who are willing to review the computer code. [Economist, March 31, 2005]

1 Introduction

Software verification research has become once again a very exciting research area and simultaneously a melting pot for many different subareas within computer science. In this position paper, I emphasize two of the areas that I am particularly interested in and consider to be of significant importance to this endeavor; meta-logical frameworks and formal digital libraries.

The central concepts in software verification are program code and formal proofs. Proofs vouch for the fact that software adheres to its specification. Specifications may be chosen arbitrarily, while within the confines designated by the specification logic. A first approximation to software correctness is software soundness. Type soundness or resource soundness guarantee that a piece of software doesn't consume too many resources (including time and space) and doesn't crash leaving the computer system in an undefined state that could be exploited by malicious intruders. Memory soundness restricts access to private and protected data. However, the task of software verification goes much beyond soundness in that it requires that any computation always adheres to its specification.

The days when software was written in only one language are long gone. In fact, a program is rarely ever a monolithic construction but almost always draws on a large variety of different functions, written in different programming languages with different semantics. These functions are organized in the form of libraries and usually invoked by name through (possibly even foreign) function

[⋆] This research has been funded by NSF grants CCR-0325808 and CCR-0133502.

interfaces. For example, the infamous *malloc* implemented in C may be called upon by a compiler written in SML.

Correctness proofs, on the other hand, still tend to be monolithic constructs. This is because, logically speaking, it is really not clear in general how specification logics interact. For example, what does it mean to appeal to a lemma proven in constructive type theory from within a classical argument about the correctness of a piece of code. It is not surprising that there is little sharing of libraries of proofs between systems like PVS, Isabelle, Nuprl, Coq, Lego, or Twelf. I argue that for software verification, we need not only worry about the interaction of different programming languages, but also the interaction of different specification logics.

Hence we must carefully distinguish between *libraries of code* and *libraries of proofs*.

There are numerous specification logics, including logics with special connectives for reasoning about time (temporal logics) and mutable state (separation logic). I prefer to think about the meaning of logical propositions proof-theoretically (as opposed to set theoretical or model theoretical), because it often yields a well-understood syntactic way of handling semantics. Those syntactic denotations are also called *proof objects* that can be independently verified and capture the evidence why a proposition holds. Syntactic proof objects have become very popular in the last decade, witnessed by the design of numerous proof carrying code architectures where explicit proof objects are shipped from code producers to code consumers as evidence that the obtained byte code adheres to an a priori defined safety policy. In such an architecture it is the compiler that needs to construct the proof object, possibly drawing on libraries of proofs.

One way of giving the machine access to abstract concepts such as meaning or derivability is by capturing the defining judgments as types in a logical framework. For example, in the logical framework LF [HHP93], the judgment that "an expression e evaluates to a value v" is encoded as a type $(\mathsf{eval}\ \ulcorner e \urcorner\ \ulcorner v \urcorner)$, where $\ulcorner e \urcorner$ corresponds to the encoding of expression e and $\ulcorner v \urcorner$ as an encoding of value v. Similarly, the judgment that F is derivable in HOL may be captured by a type $(\mathsf{hol}\ \ulcorner F \urcorner)$ where $\ulcorner F \urcorner$ is an encoding of formula F.

In the judgments as types paradigm, derivations are subsequently encoded as objects of the corresponding type. For example, HOL proof objects are encoded as objects of type $(\mathsf{hol}\ \ulcorner F \urcorner)$. The encoding is said to be *adequate* if type-checking automatically entails the validity of a derivation. Over the years, many different and more expressive logical frameworks have been developed with one goal in mind: to sharpen the conciseness of encodings. Please consult [Pfe99] for a survey article.

Thus, software verification goes far beyond one programming language and one specification logic. The effort must be seen holistically, which involves integration of different semantic programming models and the validity of mathematical knowledge beyond the borders of a particular specification logic. I believe it is here where we will find many challenging and interesting research problems in the future.

2 Challenges

2.1 Semantic Modeling

Logical framework technology has provided a reliable representation methodology for a substantial class of programming languages, including the semantic characterization of machine code [Cra03] and a complete encoding of the full semantics for SML (an ongoing project led by Robert Harper and Karl Crary)[1]. There is a large pool of practical programming languages, whose semantical foundations may mathematically not be as clean and as well understood as those of Haskell, yet need to be urgently looked at as well. Scripting languages and query languages form the basis of modern web based computing and must play a central role in any serious verification effort.

And similarly, logical framework technology has proven useful for studying properties of specification logics, including soundness and completeness properties, yet there are other specification logics that have not been looked at, which may influence the design of future logical frameworks. The current state of the art logical framework technology can capture many derivations from higher-order logic to sub-structural non-commutative logics, yet little research has been conducted on how to model modal and temporal logics. Those logics are of particular interest, especially for specifying concurrent programs.

An important challenge arises when proof theoretic means are not enough to capture meaning. In contrast to searching for proof objects, decision procedures that are used increasingly within software verification tools guarantee adherence to a specification by executing a program. Model checkers, for example, traverse large state spaces in the search of *bad* states. Combining the dynamic aspects of decision procedures with the static aspects of deductive reasoning, I believe, is an exciting and important research challenge that is still in its infancy.

2.2 Morphisms

The act of verifying software requires not only a solid understanding of the meaning and behavior of programs written in a single programming language but also the intricate details of the interaction with programs written in other languages. Analogously, proofs need to be understood not only in the context of a single specification logic, but interpreted instead in some desired target logic, be it constructive type theory or separation logic.

I think about a software verification infrastructure as the integrated whole of several programming languages and specification logics together with a set of morphisms between them as indicated by the edges in Figure 1. It is useful to distinguish between *vertical morphisms*, i.e. morphisms that express semantic embeddings among programming languages or logics, and *horizontal morphisms*, i.e. morphisms that encode programming languages and their semantics inside a specification logic. Vertical morphisms are denoted by undirected edges among

[1] Personal communication.

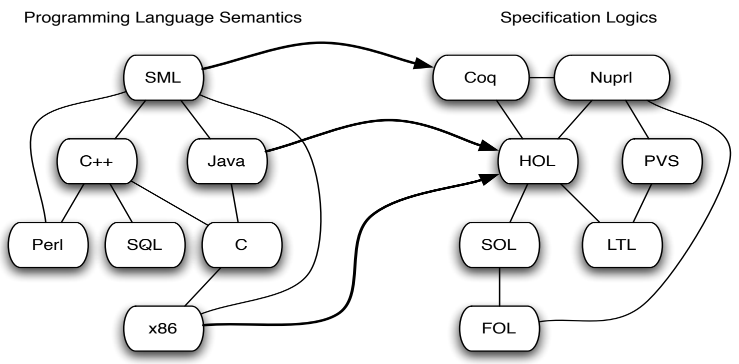

Fig. 1. Vertical and Horizontal Morphisms

programming languages and logics and horizontal morphisms by directed edges between programming languages and logics.

Vertical Morphisms. Consider for example the embedding of HOL into Nuprl pioneered by Howe [How96,How98]. Every theorem and every proof in HOL can be translated into a Nuprl proof. The meta-theoretical argument is constructive and therefore defines a total function for proof translation [SS05]. This function is a *vertical morphism* as it expresses a relation between two specification logics. The judgment that an HOL formula A can be translated into Nuprl formula B is captured by the type $(\mathsf{trans}\ \ulcorner A \urcorner\ \ulcorner B \urcorner)$ where $\ulcorner A \urcorner$ and $\ulcorner B \urcorner$ are encodings of the respective formulas in LF. Not every morphism needs to be total. To keep proof translation practical, morphisms may need to remain partial in oder to translate more expressive specification logics into less expressive ones.

Programming morphisms involves programming with meanings and proof objects. In my group, we are pursuing the development of logical framework specific programming languages for programming (partial) morphisms. The programming language whose term algebra is that of the logical framework LF is called Delphin.

Example 1 (Logic embeddings). Without going into the details over how to write Delphin programs, we give the sample Delphin type of a function f that converts HOL proofs into Nuprl proofs of the translated HOL statement. Let A be the HOL sentence, and H the proof of A in HOL. We can show that there exists a Nuprl sentence B, evidence E that B is in fact the translation of A, and of course the Nuprl proof N of B. In Delphin, we carefully separate between data expressed in the logical framework LF and programs. If $\ulcorner A \urcorner$ is the LF representation of HOL formula A, $\langle \ulcorner A \urcorner \rangle$ stands for the corresponding embedded value in Delphin. Similarly, $\langle \mathsf{hol}\ \ulcorner A \urcorner \rangle$ is an (atomic) Delphin type that merely states that A is provable in HOL.

$$f\ \langle\ulcorner A\urcorner\rangle\ \langle\ulcorner N\urcorner\rangle = (\langle\ulcorner B\urcorner\rangle, (\langle\ulcorner E\urcorner\rangle, \langle\ulcorner N\urcorner\rangle))$$

In Delphin we write $\supset$ for non-dependent function types, Π for dependent ones, and $\star$ and Σ for Cartesian products. Thus, we derive that

$$f \in \Pi\langle A\rangle \in \langle o\rangle\,.\langle \mathsf{hol}\ A\rangle \supset \Sigma\langle B\rangle \in \langle o\rangle\,.\langle \mathsf{trans}\ A\ B\rangle \star \langle \mathsf{nuprl}\ B\rangle.$$

The most noteworthy feature of Delphin's dependent type constructors is that binding occurrences of variables are determined by pattern matching, such as $\langle A\rangle$ and $\langle B\rangle$. □

Vertical morphisms also surface in the realm of programming languages in form of compilers. Proving properties about a high-level program is of little use when compiling it with a buggy compiler. It is therefore essential that the compilation process is semantics preserving.

Example 2 (Compilers). Let S be a valid program written in SML with meaning V and let us assume that this relation can be formally encoded in LF as $(\mathsf{sml}\ \ulcorner S\urcorner\ \ulcorner V\urcorner)$. Similarly, let us assume we can capture the meaning of an intermediate language program M as W, where we encode the relation between M and W in LF as a type $(\mathsf{il}\ \ulcorner M\urcorner\ \ulcorner W\urcorner)$. Furthermore, let us specify a compiler as a relation between SML programs and intermediate code, which is encoded in LF as type $(\mathsf{compile}\ \ulcorner S\urcorner\ \ulcorner M\urcorner)$. Finally we encode an interpretation function of machine language values into SML as LF type $(\mathsf{interpret}\ \ulcorner W\urcorner\ \ulcorner V\urcorner)$. A compiler is semantics preserving, if we can write a program c in Delphin such that

$$c\ \langle\ulcorner S\urcorner\rangle\ \langle\ulcorner V\urcorner\rangle\ \langle\ulcorner D\urcorner\rangle = (\langle\ulcorner M\urcorner\rangle, (\langle\ulcorner W\urcorner\rangle, (\langle\ulcorner C\urcorner\rangle, (\langle\ulcorner E\urcorner\rangle, \langle\ulcorner I\urcorner\rangle)))),$$

and show that c is total. D is evidence that V is the meaning of S, C is evidence that S compiles into M, E is evidence that the meaning of M is indeed W, and I establishes the relation between W and V. In Delphin, c has type

$$\begin{aligned}&\Pi\langle S\rangle \in \langle exp\rangle\,.\Pi\langle V\rangle \in \langle value\rangle\,.\langle \mathsf{sml}\ S\ V\rangle\\ &\supset \Sigma\langle M\rangle \in \langle mcode\rangle\,.\Sigma\langle W\rangle \in \langle answer\rangle\,.\\ &\quad\langle \mathsf{compile}\ S\ M\rangle \star \langle \mathsf{il}\ M\ W\rangle \star \langle \mathsf{interpret}\ W\ V\rangle.\end{aligned}$$

□

Horizontal Morphisms. In specification logic, programs and their meaning are the central subjects of reasoning. Since we propose to formalize both programming languages and specification logics in the same logical framework, it is important to examine how to embed programming languages into the term algebra of the logic. This embedding is made explicit by horizontal morphisms that translate programs and their meanings into the logical domain and make them available for radical scrutiny.

Example 3 (Greatest Common Divisor). The following SML program

$$\begin{aligned} \textbf{fun}\ \mathrm{gcd}\ a\ b = {} & \textbf{if}\ a\ \textbf{mod}\ b = 0\ \textbf{then}\ b \\ & \textbf{else}\ \mathrm{gcd}\ b\ (a\ \textbf{mod}\ b) \end{aligned}$$

can be represented as an object of type (exp) in LF. Recall that in LF, we write ($\mathsf{eval}\ \ulcorner e \urcorner\ \ulcorner v \urcorner$) for the encoding of the operational meaning of programs. The property we would like to prove in the specification logic is that if $\mathrm{gcd}(a, b)$ returns c, that c is indeed the greatest common divisor of a and b, which means that c divides a and b, and every other divisor of a and b is also a divisor of c.

The proof proceeds by induction on the structure of the derivation that $\mathrm{gcd}(a, b)$ returns c. Therefore, for the formal argument to be carried out in Coq, we need to turn the definition of "being an expression" and the "operational meaning" into inductively defined types and make them available to the specification logic by means of reflection.

For example, the fact that a is a valid expression is expressed by an inductive type exp providing a new infix constructor @ for application, and that v is the semantic value of e is encoded as an inductive type as well, $\mathsf{eval}\ \ulcorner e \urcorner\ \ulcorner v \urcorner$.

$$\begin{aligned} & \forall (a : \mathsf{exp}).\forall (b : \mathsf{exp}).\forall (c : \mathsf{exp}).\mathsf{eval}(\ulcorner gcd \urcorner\ @\ a\ @\ b)\ c \\ & \qquad \supset (c \mid a) \wedge (c \mid b) \wedge (\forall (d : \mathsf{exp}).(d \mid a) \wedge (d \mid b) \supset (d \mid c)). \end{aligned}$$

□

A *meta-logical framework* extends a logical framework with tools for programming and reasoning with deductive systems, including programming languages and specification logics together with a collection of vertical and horizontal morphisms. Reflection is only one of the challenges that arises when designing a meta-logical framework as the foundation of a software verification infrastructure. A reflective meta-logical framework that supports standard induction principles was described by Basin and Meseguer [BCM04]. However, it remains yet to be investigated on how to extend their result to accommodate non-standard induction principles, which are prevalent implicitly in (almost all) LF encodings. Therefore, I consider research on meta-logical frameworks, in general, an important part of the grand challenge for software verification.

2.3 Organization of Mathematical Knowledge

Many verification tasks rely on domain-specific knowledge. For example, numerical applications may have to appeal to properties of floating point arithmetic or even stability results for solutions of sets of differential equations. But what shall one do, if the specification logic is Isabelle/HOL yet all desired theorems were already proven in PVS? It seems to be a waste of effort to redo all those parts from scratch. Thus, I have become very interested on how to share mathematical knowledge and proofs across logic specific boundaries.

In a *formal digital library* we store a collection of mathematical facts together with their respective meanings in form mathematical proofs. Vertical morphisms help cast mathematical knowledge from one formalism into another. A formal digital library should be extensible, meaning that new logical systems and vertical morphisms can be easily added. Furthermore, it should provide adequate functionality to retrieve and maintain mathematical knowledge in a database.

I speculate that such a formal digital library is not only applicable for libraries of proofs, but that it also helps organize libraries of code. Moreover, if invariants and their proofs are stored in conjunction with code, the query engine could use this information and retrieve library functions, not by name, but by a semantic description.

Example 4 (Relative primality). Assume that a programmer tries to write a function to compute if two numbers a and b are relatively prime. He or she might formulate the following (speculative) query: Is there a program S in SML, that has the following invariant provable in Coq?

$$\forall a : \mathsf{exp}.\forall b : \mathsf{exp}.\forall c : \mathsf{exp}.(c \mid a) \wedge (c \mid b) \supset c = 1$$

The answer should be: Yes, choose

$$S = (\mathrm{gcd}\ (a, b) = 1\ \textbf{andalso}\ \mathrm{gcd}\ (b, a) = 1).$$

$\square$

At least two different formal digital library projects are currently underway. The FDL project led by Cornell University, the California Institute of Technology, and the University of Wyoming focuses on the organization of mathematical knowledge. The Logosphere project conducted at Carnegie Mellon University, SRI International and Yale University concentrates on the organization of libraries of mathematical proof and the specification of vertical morphisms.

3 Conclusion

The enormously difficult task of verifying software involves a colorful variety of programming languages and specification logics and is intimately connected to mathematical truth. Sometimes this truth can be retrieved, other times it must be constructed. It is common knowledge that the performance of verification technologies depends on the kind of verification problems it is applied to, and each technology has its weaknesses and its strengths. The more we can draw on all areas of automated deduction and formal methods, the faster we can reach the goal of software verification. Thus, a common meta-logical framework infrastructure that facilitates the construction of libraries of code and libraries of proofs will help us make strides towards a solution of the grand challenge.

Returning to the original opening quote to this position paper, a formal digital library may help Dr. Hales to completely formalize the proof of Kepler's conjecture in HOLlight, his chosen target logic. The vertical morphisms will help to convert pieces of the proof formulated in different logical formalisms and the horizontal morphisms allow him to reason about the correctness of the programs he used to obtain the proof, leading up to one huge monolithic proof object in HOLlight, that can be subsequently checked by a small and well-understood proof checker. Trusting the checker means trusting the proof, which simplifies the job of the referee who does not even have to look at the proof at all any more.

References

[BCM04] David Basin, Manuel Clavel, and José Meseguer. Reflective metalogical frameworks. *ACM Transaction on Computational Logic*, 5(3):528–576, 2004.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th Symposium on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisiana, USA, 2003.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[How96] D. J. Howe. Importing mathematics from HOL into Nuprl. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282. Springer Verlag, 1996.

[How98] D. J. Howe. Toward sharing libraries of mathematics between theorem provers. In *Frontiers of Combining Systems, FroCoS'98, ILLC, University of Amsterdam, October 2–4, 1998, Proceedings*. Kluwer Academic Publishers, 1998.

[Pfe99] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.

[SS05] Carsten Schürmann and Mark-Oliver Stehr. An executable formalization of the HOL/Nuprl connection in the meta-logical framework Twelf. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Montego Bay, Jamaica, 2005. Springer Verlag. to appear.