Programming with Proofs: Language-Based Approaches to Totally Correct Software

Aaron Stump

Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA

1 Introduction

Tremendous progress has been made in automated and semi-automated verification since the seminal works on program verification. Automated deductive techniques like model checking have been highly successful for many verification tasks (e.g., [17, 18, 13]). Impressive advances continue to be made in static analysis, type systems, and static bug finding (e.g., [21, 12]). These approaches aim to verify code or find bugs in existing systems as automatically as possible, with as little developer help as possible. This has been the aim of the research community for many years, possibly due in part to the bad reputation that continues to plague full program verification. Theorem proving approaches to program verification have continued to make advances, but indeed, they still are generally applied only to the most critical applications (e.g., [7, 5, 16, 11]).

Despite the continuing advances in fully automated verification, it seems unlikely that essentially automatic techniques will ever be able to scale to full program verification. Given steadily increasing societal reliance on software systems, totally correct code remains a vitally important goal. In this position paper, I advocate an approach to full program verification in which programmers write imperative programs and their computational proofs together as single artifacts (Section 3). Despite the reliance on manual creation of proofs, the approach I advocate is quite different from existing theorem-proving approaches, which I argue are unlikely ever to be feasible for mainstream use (Section 2). In Section 4, I show how the approach I advocate solves critical problems with theorem proving, and I compare the approach to other verification approaches.

2 Problems with Theorem Proving

Program verification based on theorem proving deserves its unfavorable reputation as unreasonably burdensome. Consider a typical contemporary example, the Krakatoa tool for certifying Java and JavaCard programs [8]. The approach implemented in this tool is state-of-the-art, and integrates many sophisticated ideas and tools to provide a complete solution for an important real-world verification problem. Nevertheless, despite its impressive strengths, Krakatoa demonstrates the burdensome nature of current theorem-proving approaches. Krakatoa works

as follows. Verification problems consisting of Java programs annotated with JML specifications are translated into verification problems in an intermediate language called WHY. Another tool then generates verification conditions (VCs) for the WHY problem, in the language of the Coq proof assistant. Those VCs must now be proved in Coq. While Coq provides a number of powerful tactics for automated reasoning, this task is of necessity largely a manual one. The proof is conducted with respect to a model in Coq of parts of Java carried through by the WHY tool. The number of highly complex artifacts that the person doing the verification must understand here is simply too great. She must be fluent in:

- the specification language, in this case JML. JML's syntax resembles Java's, but as a logical language it nevertheless relies on mathematical, as opposed to computational, intuitions.
- the proof language, in this case the language of Coq. As a higher-order logic, this is far removed from something most developers are familiar with. Effective verification in a tool like Coq also requires knowledge of a sophisticated tactics library.
- the background theory; in this case, the partial model of Java in Coq.
- in the case of Krakatoa, the WHY intermediate language, and the encoding of Java into WHY. This must be understood since the VCs are generated via WHY. The encoding of Java into WHY is nontrivial, involving, for example, an explicit model of the heap.

It is not reasonable to require a programmer to understand all these unfamiliar artifacts to solve even basic verification problems.

3 Language-Based Verification

Program verification based on proving extracted verification conditions (I will call this "the VC approach") seems destined to remain infeasible for mainstream use. But there is another approach, known for a long time to the type theory community, which does not suffer from these difficulties. This is what Thorsten Altenkirch calls *internal verification* [2]. With internal verification, proofs are data in the programming language, just like booleans or strings. The type of a proof is the theorem it proves. Internally verified functions require, as additional input arguments, proofs of their pre-conditions. They produce, as additional outputs, proofs of their post-conditions. Proofs are connected to the non-proof parts of programs by the type checker. My group[1] has been developing two different languages based on this idea, which I will now describe. The first is nearing a certain degree of maturity, while the second is still in very early stages.

3.1 RSP1

RSP1 is a novel functional programming language that supports type-safe imperative programming with proofs [20]. It builds on ideas from Martin-Löf type

[1] See http://cl.cse.wustl.edu.

theory, particularly as developed in the logical frameworks community, while adding imperative features. The basic idea of logical frameworks is that proof systems can be encoded as term-indexed datatypes. For example, we can declare a datatype `form` of formulas in a standard way. Then a datatype `pf` of proofs may be declared as a term-indexed datatype, where the index is a formula. If `p` is an encoded formula, then `(pf p)` is a type. Term constructors are then declared in such a way that mathematical proofs of formulas ϕ are in one-one correspondence with values of type (`pf phi`), where `phi` is the object of type `form` corresponding to ϕ. For example, we might have a constructor `ModusPonens` which takes in formulas `p` and `q`, together with objects of type `(pf (implies p q))` and `(pf p)`, and constructs a new term of type `(pf q)`. As this example shows, the types of constructors are *dependent* in RSP; the return type of a use of `ModusPonens` depends on (i.e., mentions by name) some of its input values. The static and dynamics semantics of RSP1 have been formalized, and the language has been proved type safe. The major difference between RSP1 and Martin-Löf type theory is that the latter relies on evaluating arbitrary terms at compile-time to doing type checking. This means that imperative features cannot be added to Martin-Löf type theory in any straightforward way, and programs must all be (strongly) normalizing. RSP1 separates representation and computation in such a way that imperative features and general recursion can be handled, while retaining decidable type checking. The essential technical idea is syntactically to keep impure constructs like effectful operations out of types. To refer in a type to something like the result of a possibly non-terminating computation or a read of mutable state, one uses a *hiding let* construct to get a name for the result of the computation. This name can then be used in types. An RSP1 type checker, interpreter, and compiler to Ocaml have been implemented, in around 7000 lines of Ocaml.

We have implemented a number of nontrivial examples in RSP1. One is a proof-producing validity checker called RVC ("Rogue Validity Checker"), which is currently around 8000 lines of RSP1 (see [6] for a description of RVC in its early stages). RVC decides validity of quantifier-free formulas modulo combined background theories of linear integer arithmetic, uninterpreted functions, and arrays. Due to type safety of RSP1, any proof produced by a successful execution is guaranteed to check. Indeed, we have implemented a form of partial evaluation which can slice out proofs from RVC's code after type checking. Other examples include statically validated mesh-manipulating algorithms from Computer Graphics, where internal verification ensures that the data being manipulated always satisfy the property of being a mesh (with a particular Euler characteristic) [3].

3.2 Local Heap Invariants in RSP1

RSP1 can be used to write programs with proofs showing that local invariants of the heap are maintained. This can be done even though RSP1's type system does not allow explict mention of the results of reads and writes to mutable state in types (which serve as specifications). In RSP1, pointers can be set to point

from one object to another. To express a local invariant, the programmer can require that a proof of that invariant must be given when the pointer is set. Such a proof is then available when the pointer is dereferenced. Technically, this is done by making the pointer point from the first object to a *dependent record* (as in [15]) containing the second object and the proof of the invariant.

Local heap invariants, while less expressive than global invariants involving, for example, the transitive closure operator, are still quite powerful for specifying mutable data structures. Several examples are explored in a recent paper from my group, the most complex of which is verified insertion into a binary search tree, where we statically verify the binary search tree property [20]. This is done by associating, with each node in the tree, a lower and upper bound for all the data at nodes reachable from the current node. This association is done using an indexed type: nodes have type "`node l d u`", where `l` is the lower bound, `u` is the upper bound, and `d` is the data stored at the node itself. Setting the pointer to the left subtree of a node requires a proof that the upper bound of the left subtree is less than or equal to the data stored at the current node (and similarly for the right subtree). Insertion then manipulates proofs showing that these local invariants hold. Verification using local heap invariants has also been studied recently by McPeak and Necula [10].

3.3 Reflected Evaluation Proofs

RSP1 is excellent for programming with proofs of properties of data. It can even verify certain properties of mutable data structures, despite the fact that types are forbidden to mention references. But RSP1 is not well suited to verifying total correctness of algorithms, since it provides no way to reason about executions of code. For example, it has no mechanisms to prove that code will not encounter run-time errors like arithmetic exceptions or array-bounds errors. Furthermore, specifications are implemented as datatypes, and proofs must be defined by the programmer.

A more fundamental approach to programming with proofs may possibly be realized along the following lines. First, we adopt assertions as our (computational) specifications. That is, pre- and post-conditions of functions are expressed by saying how certain pieces of code are expected to evaluate before and after the function is called. For example, consider the obvious function which merges two sorted lists to obtain a new sorted list with exactly the same elements. Part of its computational specification is that calling the `check_sorted` routine on each input list before the function is called should return true. This way of specifying code is likely to be very appealing to programmers, who are already used to writing assertions. But the question then arises, if we are going to program with proofs for these kinds of specifications, what are those proofs? The answer is that they are reflected versions of the evaluation proofs inductively defined by the operational semantics of the language. For each proof rule in the operational semantics of the language, there is a built-in term constructor. Reflected evaluation proofs showing that pieces of code execute in certain ways can then be built by the programmer in the programming language. The type of the proof is the

evaluation statement it proves. For example, in the big-step semantics for a language with an if-then-else (ITE) construct, there is a rule that says: if the if-part of the ITE evaluates to true, and the then-part evaluates to X, then the whole ITE evaluates to X. This proof rule is reflected as a term constructor `iftrue` in the programming language. This constructor takes in two reflected proofs corresponding to the two premises of the rule, and constructs a term whose type is `(if I then T else E ==> X)`.

The foundational nature of reflected evaluation proofs is appealing, but it is not clear yet how the technical development of the idea should proceed. One approach begins by defining static and dynamic semantics for such a language, and proving the usual meta-theoretic results, in particular type preservation. One technical novelty is that whenever a term is evaluated, the operational semantics states that the corresponding reflected evaluation proof is actually constructed, and placed on a stack. Subproofs are removed from this stack to construct the new proof. These proofs are then accessible to programs via an explicit reflection construct.

A second approach begins by developing a logical theory of executions of a programming language (without embedded proofs). Executions are represented by terms in some logic, most likely higher-order logic, for handling binding constructs in the programming language. Axioms are given defining a relation on executions and pairs of terms, which holds when the first term evaluates to the second according to the execution. An induction principle is formulated for executions. Properties of programs may then be proved in the external verification style using higher-order logic and the axioms about executions. To support internal verification, the programming language is then extended to allow executions as program data. The meaning of a program is partly determined by an elaboration function which maps a program (with its embedded proofs in the internal verification style) into a higher-order logic proof about the computational part of the program (i.e., without its embedded proofs). The goal is then to define a type system which is sound for successful elaboration: well-typed programs elaborate to higher-order logic proofs that are guaranteed to check. The advantage of this approach over the first one is that it naturally supports classical reasoning principles. The first approach seems most naturally to require constructive reasoning.

Whichever approach to the technical development of reflected evaluation proofs succeeds, techniques for proof irrelevance will likely be required to slice away proofs from the computational parts of programs [14].

4 Comparison to Existing Approaches

I will now argue that the two languages described in the preceding Section compare favorably with existing verification approaches, using these metrics:

automatic: How automatic is the approach? Must the programmer write proofs, specifications, or other annotations, or can raw code be handled?

strength: Can arbitrary properties be specified and checked of arbitrary systems, or are the specifications or systems that can be handled restricted in some way?

mainline languages: Does the approach apply to systems developed in mainstream programming languages, or must special languages be used?

incremental update: How much work is required to re-establish specifications after an incremental change?

unified language: Is there a single language and single set of tools for specification, implementation, and proof?

computational approach: Are the languages involved computational or mathematical/logical in character?

support for imperative features: Can the approach handle imperative features? If so, how directly are they handled?

Figure 1 compares the following approaches with the languages proposed above: fully or mostly automated approaches like model-checking and static analysis, theorem-proving based on manually proving extracted VCs, program development by refinement, and type theoretic approaches like Martin-Löf type theory (MLTT in the Figure). The latter are, of course, the most closely related to RSP1 and the reflected evaluation proofs (REP in the Figure). The KeY system supports verification of JavaCard programs using Dynamic Logic [1, 4]. Let me provide some further explanation for the entries in the Figure.

approach	**auto**	**strong**	**mainline**	**update**	**unified**	**comp.**	**effects**
automated	yes	no	yes	yes	partial	partial	yes
VCs	no	yes	yes	partial	no	no	yes
refinement	no	yes	yes	no	no	no	yes
MLTT	no	yes	no	yes	yes	yes	no
RSP1	no	partial	maybe	yes	yes	yes	partial
REP	no	yes	maybe	yes	yes	yes	yes(?)
KeY	partial	yes	yes	yes	good	partial	yes

Fig. 1. Comparison of verification approaches

Automated approaches cannot handle arbitrary properties of arbitrary systems. Specifications (e.g., temporal logic in model checking) are often more logical than computational. The techniques do apply to mainstream programming languages, and can handle effects. The VC approach has already been discussed. Note that while it is in principle possible to apply the VC approach to any programming language one likes, doing so for a new language typically requires a lot of modeling work in the theorem prover. Refinement requires manual application of refinement steps, which are outside the programming language. The main drawback of the approach of Martin-Löf type theory is that it cannot in any obvious way be extended to mainstream programming languages with general recursion or effects. RSP1 goes some of the way towards dealing with effects

in a unified, computational way. The ideas there could probably be incorporated into a mainstream functional programming language like Ocaml. It does require manual creation of proofs, like the VC approach. I do not consider it fully strong because of its limited ability to specify properties of the heap. The reflected evaluation proofs approach has the same strengths as RSP1, with the added potential (not yet realized) to specify properties of the heap. Furthermore, reflected evaluation proofs support an even more computational approach than RSP1, since specifications are just assertions, whereas in RSP1, they are term-indexed datatypes.

The KeY system, while based on a traditional verification paradigm (namely Dynamic Logic), supports a much more viable approach to dispatching verification conditions than is typical of the VC approach. In particular, program fragments are retained in the modal operators of Dynamic Logic, and discharging verification condition requires a mix of symbolic simulation, state simplification, and logical deduction. In this respect, it resembles work on Dynamic Verification [19]. KeY also has some support for incrementally recovering parts of proofs which are still applicable when code is modified. Proofs are separate artifacts from programs, but the KeY tool is able to maintain a close connection between proofs and programs. It remains to be seen how the tightly integrated language-based approach I have advocated here compares in detail with an approach like KeY's.

5 Conclusion

I have argued that traditional approaches based on proving extracted verification conditions are unlikely ever to be widely adopted for verification, due to the heavy burdens they place on developers. I propose that instead, language-based approaches to program verification be developed, where correctness proofs are intertwined with code, in a way that gives them the appearance of being program data just like strings or booleans. Work in this direction is being pursued by my group, based on the RSP1 language, and the still developing idea of reflected evaluation proofs. The intention is that by making specificational and verificational artifacts more computational and less (in the technical sense) logical, program verification will become more usable for regular development. Other projects working toward this vision include the Epigram project and the ATS project [22, 9]. Language-based approaches are also highlighted at the FLoC 2006 workshop entitled "Programming Languages meets Program Verification" (PLPV), organized by Hongwei Xi and the author.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

2. Thorsten Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. Available from the author's website.
3. Joel Brandt. What a Mesh: Dependent Data Types for Correct Mesh Manipulation Algorithms. Master's thesis, Washington University in Saint Louis, April 2005. Available from `http://cl.cse.wustl.edu`.
4. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
5. J. Harrison. Formal Verification of IA-64 Division Algorithms. In *13th International Conference on Theorem Proving in Higher Order Logics*, 2000.
6. R. Klapper and A. Stump. Validated Proof-Producing Decision Procedures. In C. Tinelli and S. Ranise, editors, *2nd International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.
7. G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.
8. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa Tool for Certification of JAVA/JAVACARD Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.
9. C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.
10. S. McPeak and G. Necula. Data Structure Specifications via Local Equality Axioms. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer-Aided Verification*, pages 476–490. Springer, 2005.
11. J. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5k86 Floating-Point Division Program. *IEEE Transactions on Computers*, 47(9), 1998.
12. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.
13. G. Norman and V. Shmatikov. Analysis of Probabilistic Contract Signing. In *In BCSFACS Formal Aspects of Security (FASec '02)*, 2002.
14. F. Pfenning. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *16th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001.
15. Robert Pollack. Dependently Typed Records in Type Theory. *Formal Aspects of Computing*, 13:386–402, 2002.
16. H. Rueß N. Shankar, and M. Srivas. Modular Verification of SRT Division. *Formal Methods in System Design*, 14(1), 1999.
17. I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging Overconstrained Declarative Models Using Unsatisfiable Cores. In *18th IEEE International Conference on Automated Software Engineering*, 2003. received best paper award.
18. M. Velev and R. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, February 2003.
19. C. Wang and D. Musser. Dynamic Verification of C++ Generic Algorithms. *IEEE Transactions on Software Engineering*, 23(5):314–323, 1997.
20. E. Westbrook, A. Stump, and I. Wehrman. A Language-based Approach to Functionally Correct Imperative Programming. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, 2005.
21. Y. Xie and A. Aiken. Scalable Error Detection using Boolean Satisfiability. In M. Abadi, editor, *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.

22. D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97. Springer, 2005.